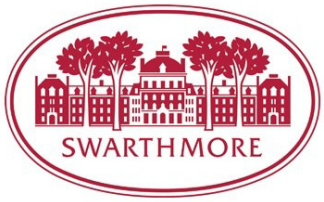

Swarthmore College
Department of Linguistics
Bachelor's Thesis

Tokenization of Japanese Text Using a Morphological Transducer

Clare Hanlon

December 13th, 2017

Project code:
https://github.com/chanlon1/tokenisation/

Created with tools provided by:
Wei En
https://svn.code.sf.net/p/apertium/svn/branches/tokenisation/
Darbus Oldham
https://wikis.swarthmore.edu/ling073/User:Doldham1/Final_project

---

Abstract

Word segmenters comprise a vital step in the methodology of natural language processing. In languages such as English, which already necessitate word delimiters such as spaces, this task is trivial. However, in non-segmented languages such as Japanese and Chinese, a translator must accurately identify every word in a sentence before or as they attempt to parse it, and to do that requires a method of finding word boundaries without the aid of word delimiters. Much has been done in this field for the case of Chinese, as Chinese is a highly isolating language which makes the task of identifying morphological units almost isomorphic to the task of identifying syntactic units. As such, many functional Chinese Word Segmenter models already exist. But

Japanese, on the other hand, is a synthetic language that utilizes both inflectional and agglutinative morphology, and so the tasks of identifying morphological units and syntactic units are more separate. However, much work has also been done in the field of mapping inflected Japanese words to their root form, a process known as transduction. In this paper, I modify an existing Chinese Word Segmenter to incorporate an existing Japanese transducer into its segmentation process: specifically, the transducer's ability to detect the validity of a combination of characters is used in parallel with dynamic programming's ability to compute all possible combinations of characters in a string to find the overall number of valid tokens in a given input string. Testing shows that this approach does indeed give valid results; furthermore, its ability to return information about the grammatical tags of each token suggests that further extensions of the program could not only tokenize the text, but also infer information about its syntactic meaning in the clause.

---

1 Introduction and Motivation

Japanese poses an interesting problem for word segmentation, as it does not utilize spaces to delimit boundaries between words, but still has a concept of words. This concept is reflected by several of the more unique properties of the Japanese language: postpositions, or "particles", which mark grammatical case; *kanji* borrowed from Chinese characters for word roots; and both inflectional and agglutinative morphology to decline verbs. With such a complex grammatical scheme, a tokenizer that is able to take into account all the grammatical variations of Japanese is essential. A brief overview of the aforementioned sections of Japanese grammar will be provided in this section, summarized from *The Languages of Japan* by Masayoshi Shibatani.

1.1 Orthography

Japanese is known for having three distinct orthographies: *kanji*, *hiragana*, and *katakana*. The latter two are occasionally grouped into the same category, *kana*. The first system, *kanji*, came from the Chinese orthography of *hanzi*. The Japanese language borrowed this Chinese character system as its first written form; Japanese words were assigned to Chinese characters based on similarity in meaning. As such, every Japanese *kanji* has two different readings: its *kunyomi*, which is its Japanese pronunciation, and its *onyomi*, an adapted version of its original Chinese pronunciation. In general, *kunyomi* is used when a character appears as the root of the form, such as a single-character noun or as the stem of a verb, whereas *onyomi* is used when a character appears in compound with another character. This difference in pronunciation seems to act as a very loose word delimiter, as demonstrated by the following examples:

1.) 火山が　　噴火　　した。
kazan-ga　　*funka*　　*sita*
volcano-NOM　erupt.ABS　do.PST.IFML
"The volcano erupted."

2.) 大火　　山まで　　ついた。
daika　　*yama-made*　　*tsuita*
large fire.ABS　mountain-ABL　reach.PST.IFML
"The large fire reached the mountains."

Fig. 1: Two different occurrences of the characters 火 and 山 in conjunction; 山 in the first is read using its *onyomi*, pronounced "*zan*", while the second example uses its *kunyomi*, pronounced "*yama*".

In the first example, 火 and 山 are compounded together to make one word, 火山, which means volcano. 火 is read with its *onyomi*, "*ka*", and 山 is also read with its *onyomi*, "*zan*". In the second example, however, 火 is compounded with the preceding character 大 to make a different word meaning "large fire" . So while 火 still uses its *onyomi* here, 山, which is not compounded with any other *kanji*, uses its *kunyomi*, "*yama*". In this way, the different readings of *kanji* indicate whether they are a standalone word or part of a compound word.

The *kana*, on the other hand, only have one pronunciation for each character. Both *hiragana* and *katakana* represent the same sounds, as both were derived from *kanji* that previously represented that sound. Over time, this character set was simplified into the two existing *kana* systems that are used today. *Hiragana* is used for thing such as particles, simple words, and auxiliary verbs, while *katakana* is mostly for foreign loan words.

For a final overview, observe the following example sentence that uses all three orthographies:

2.) テレビを　　見よう。
terebi-o　　*miyou*
television-ACC　watch-VOL.IFML
"I'm going to watch TV."

Fig. 2: Example sentence using all three Japanese orthographic systems.

The first word, which comes from the English word "television", is written in *katakana*, as it is a borrowed word. The particle that follows it is written in *hiragana*. The stem of the verb, 見, is the *kanji* for "see/watch": since there are many verbs with the stem "*mi*" in Japanese, the orthography is capable of

showing which *"mi"* this is. Finally, the inflectional ending is written in *hiragana,* since it is assumed that the listener/reader understands this to be a piece of inflectional morphology and does not need its meaning to be specified.

This use of orthography to distinguish types of words raises the question of whether this heuristic can be used to inform the tokenization process.

1.2 Verb Conjugation

Japanese has two different types of verbs, "*U*-verbs" and "*Ru*-verbs. When conjugating either of these types, the root of the verb must be inflected to the desired form, and then, depending on that form, an auxiliary verb may follow. Japanese schools teach this concept to students using the rules expressed in the following chart:

Table 10.3. *Inflectional categories of Modern Japanese*

		'die'	'look at'	
Mizen	(Irrealis)	*si-na*	*mi*	(*m-i*)
Renyoo	(Adverbial)	*si-ni*	*mi*	(*m-i*)
Syuusi	(Conclusive)	*si-nu*	*miru*	(*m-iru*)
Rentai	(Attributive)	*si-nu*	*miru*	(*m-iru*)
Katei	(Hypothetical)	*si-ne*	*mire*	(*m-iru*)
Meirei	(Imperative)	*si-ne*	*miro/miyo*	(*m-iro/iyo*)

Fig. 3: Inflectional categories of Japanese
as defined by grammar textbooks,
from Shibatani (1990), p. 224.

The chart in Figure 3 shows examples of the two different types of verbs undergoing all potential inflectional processes. The first verb, "die", a "*U*-verb", has the stem /*sin*/, while the second verb, "look at", a "*Ru*-verb", has the stem /*mi*/.

After the verb has been inflected into the desired form, the inflectional category then dictates what auxiliary grammar items can be attached, as defined in the next table.

Table 10.4. *Subcategorization of auxiliaries and conjunctive particles according to the inflectional categories (Modern Japanese)*

Mizen:	*nai, n(u)* (Negative); *seru, saseru, reru, rareru* (Voice/Honorific)
Renyoo:	*ta* (Past); *masu* (Polite); *tai* (Desiderative); *soo* (*da*) (Conjectural); *tari, te, tutu, nagara* (Conj. particles)
Syuusi:	*rasii, soo* (*da*) (Hearsay); *to, kara, ga* (Conj. particle)
Katei:	*ba* (Conj. particle)

Fig. 4: Auxiliary words divided by the inflectional categories they can be suffixed to, from Shibatani (1990), p. 224.

So, this process of inflecting the verb stem and then adding the necessary auxiliary grammar is the traditional Japanese understanding of verb conjugation. However, linguists, both Japanese and non-Japanese alike, disagree with the approach taught in schools, and instead propose inflectional categories such as the following:

Table 10.5. *Sakuma's inflectional categories*

	'die'	'look at'
Basic form	*sin-u*	*mi-ru*
Formative form	*sin-i*	*mi-ø*
Negative form	*sin-a*	*mi-ø*
Hypothetical form	*sin-eba*	*mi-reba*
Imperative form	*sin-e*	*mi-ro*
Future form	*sin-oo*	*mi-yoo*
Determined form	*si-nda*	*mi-ta*
Suspended form	*si-nde*	*mi-te*

Fig. 5: The proposed inflectional categories of Sakuma (1936), from Shibatani (1990), p. 226.

This representation accounts for every possible inflection of Japanese verbs, whereas the elementary version restricts itself to the first six forms in this table.

This conjugation process can very easily be replicated computationally, as it follows a very structured pattern. So, even though Japanese verbs have much more agglutinative grammar attached to them than Chinese verbs, if one takes this knowledge into account, it should be comparably simple to identify inflected forms in Japanese text.

1.3 Particles

Japanese uses "particles", a type of postposition, to explicitly mark the case of a given noun phrase. While Japanese is an SOV-ordered language, particles give speakers the freedom to change up the order of words in a sentence to express emphasis. Take the following sentence:

6.) 太郎が　　次郎を　　見た。
Tarou-ga　*Jirou-o*　*mita*
Tarou.NOM　Jirou.ACC　see.PST.IFML
"Tarou saw Jirou."

Fig. 6: A sentence in the typical SOV order.

The subject, marked here with the particle が, takes the initial spot in the sentence, and is followed by the verb. The object, on the other hand, is marked by を, and appears after the subject. This sentence is a straightforward communication of the fact that Tarou saw Jirou. However, this sentence can also be restructured as:

7.) 次郎を、　太郎が　　見た。
Jirou-o　*Tarou-ga*　*mita*
Jirou.ACC　Tarou.NOM　see.PST.IFML
"Tarou saw Jirou."

Fig. 7: A valid sentence violating the typical SOV order, taken from Shibatani (1990).

This structure adds even more emphasis to the sentence, specifically to the fact that Jirou is the one who Tarou saw. The structure itself suggests that the speaker first and foremost wanted to express the fact that Jirou is the object of some action, and then afterwards added the nominative phrase to clarify what exactly that action was. The particles here help the listener understand the sentence when they may not be able to rely on word order to communicate grammatical roles.

In casual speech, particles such as が and を are often dropped, and noun phrases appear in their absolutive form (Shibatani 2009, p. 259). In this situation, the listener must rely on word order and context to understand the grammatical roles of each word. However, when particles are present, they clearly delineate both word boundaries and case. Some confusion may arise due to the fact that the characters that comprise particles are also used in other contexts, such as in the root of common words, but a tokenizer could use an understanding of

particles to inform its segmentation process, and even be extended to also find grammatical tags, either from the particles or from word order.

2 Existing Tokenization Methods

As there has been much more work done for the Chinese version of this problem, referencing Chinese models will also prove valuable in this exercise. However, the linguistic differences between Chinese and Japanese cannot be ignored. As such, this section will be split into parts: analysis of existing Chinese word segmentation algorithms, and analysis of Japanese grammar in a more abstract sense. This section will give a general overview of the Chinese Word Segmentation models, and its two prevailing methods, statistical and deterministic.

Much has been done in the way of Chinese word segmentation, as Chinese, whose syntax and morphology are almost identical, represents a more concrete version of the tokenization problem. Both Chinese and Japanese are less structurally-fixed languages in comparison to languages such as English, but Chinese is also an extremely isolating language, whereas Japanese is a combination of both isolating and agglutinative, so this difference must be kept in mind. However, because Japanese borrows its orthography system from Chinese, many issues discussing the segmentation of Chinese *hanzi* will also be helpful when thinking about the segmentation of Japanese *kanji* compounds.

2.1 Statistical Models

A vast majority of approaches to this problem propose solving it by means of statistical methods, without any regard for the language's grammatical properties. For a language such as Chinese, where every character can be considered a lone morpheme, this approach makes a degree of sense, especially given that the average word is only about two characters long (Yao and Lua 1998, p. 313).

Most of these methods rely on some sort of dictionary of known words that the program can refer to. While a dictionary is a corpus of a language and thus linguistic, to a computerized parser, the dictionary is nothing more than valid combinations of characters. Thus, when this paper refers to "linguistic models", it is referring to models that take into account the structural properties of a language. As many of the discussions of the non-linguistics methods will mention, this is inherently limited, as a dictionary must come from a corpus, and it is a hard task for a corpus to contain every single word in a language, as languages are constantly evolving and new words are born constantly (Huang and Xue 2012, p. 497).

The greedy method approach to segmentation is widely used in conjunction with the dictionary method. In the greedy method, there are two

points of decision for the coder to make: whether the algorithm should start by finding words at the beginning of the sentence, or the end, and whether the algorithm should try to find as many words as possible or as few words as possible. These are respectively named forward and reverse matching, and maximum and minimum matching. The most commonly used method is the Maximum Reverse Matching method (Wu 2011, p. 609). However, a greedy algorithm is prone to error, as its decisions are final and are not able to be updated once new information is discovered.

While the above algorithm relies on known words, there also exist algorithms that try to calculate the likelihood of certain characters comprising a word on their own. As discussed in Huang and Xue (2012), models differ based on how they decide to pair characters. One method focuses solely on the likelihood that two characters are in the same word; this model looks over its training data and calculates the probability that two characters appear next to each other in a valid word. These probabilities then serve as weights, and words are defined based on the likelihood that adjacent characters form a word. This approach works well for words that are of two-character length, since this is the most common pattern of words in the Chinese language, and thus the most statistically probable, but does not perform as well for words of other lengths.

There have been multiple attempts (Huang et al, 2012) to minimize the error of these two methods by combining them, such as measuring the possible words calculated by internal binding against their appearance in the dictionary, thereby being guided by the dictionary in that sense.

Another method similar to the internal binding method is the word boundary method. Rather than look exclusively at two given characters' likelihood of occurring in a sequence, this algorithm computes all possible segmentations of a given character sequence and assigns the segmentation that returns the most statistically likely segmentation. This method involves no dictionary look-ups, since it relies completely on probabilities learned previously from a corpus.

Yao and Lua (2009) try to combine the statistical and linguistic methods to create a non-greedy tokenizer that still relies on the relationship between two characters while also avoiding the determinism of the statistical approach, which ignores less common but occasionally correct segmentations. The algorithm focuses on finding the largest combination of characters possible, similar to the maximum reverse matching method, but it continues to look for valid words within the subset of these words, and then chooses the sequence with the highest probability. This method is inherently greedy in its design, but by considering a larger range of possible character combinations, like the word boundary method, it allows for more flexibility.

Unfortunately, all of the methods described above contain no linguistic analysis whatsoever, and rely purely on statistics learned from a given data set. While these methods may work from a theoretical point of view, they end up performing poorly (Huang et al, 2012) in the real world, as they neglect to consider the grammatical properties of the language, which could be utilized to improve the process of tokenization.

2.2 Deterministic/Linguistic Models

In contrast, within the scope of linguistically-motivated segmentation, there are several methods based around the idea that word segmentation should be a product of parsing a sentence, rather than a means of it. Li (2000) argues vehemently that for a language like Chinese without orthographic word boundaries, it is not necessary to segment the text in order to understand it – rather, by correctly understanding the text, the divisions between words then become evident. While Li spends most of his time criticizing statistical models and never proposing a model of his own, he speaks highly of hiw own creation, CPSG95, an existing model that relies purely on morpho-syntactic rules rather than statistics.

CPSG95 takes advantage of the fact that the morphology of Chinese is extremely similar to its syntax – that is, due to the extremely isolating nature of Chinese, each character can be thought of as both a syntactic unit and a morphological unit. CPSG95 breaks a clause down into its individual characters, and first looks for certain expected units, such as a verb. After finding one of these, it will then look for the units that it would expect to appear with the previous unit: if it has found a verb, for example, it would then look to see if there is perhaps a subject or object. This also works on the morphological level: if a noun suffix is found, it would then expect there to be some noun which that suffix attaches to. This process repeats until all characters in the string have been categorized, at least tentatively. It then uses these categorizations as boundaries that identify separate words.

This approach is rather antithetical to the statistical approaches discussed above, which completely ignored linguistic information and treated all characters as symbols to be paired together. CPSF95 relies exclusively on linguistic rules to intuit the words that comprise a piece of input – if one were to remove the linguistic aspects of a piece of input, this algorithm would not be able to function.

However, Wu (2011) criticizes both of these methods, arguing that by focusing solely on rules to detect word boundaries, these algorithms are ignoring the semantic aspect of word detection that humans use to complete this task. After interviewing 8 native Chinese speakers to see how they segmented text, the author concluded that Chinese is a much more "pragmatic" language than the typical Indo-European language: in other words, Chinese relies heavily on the reader's intuition and ability to grasp the context of a situation to parse text, whereas languages such as the Romance languages follow strict rules that clearly denote a word's grammatical purpose and positioning in a clause. To solve this issue, the author proposes a three-part segmentation algorithm that utilizes "knowledge-based rules" to inform its decision-making process.

The first part of this method involves making deterministic segmentations by identifying symbols such as punctuation and known affixes to find all the obvious boundaries in the given text. These partially segmented items are then

fed into part two, which uses forward, reverse, maximum, and minimum matching methods to identify all possible word combinations in a sentence. (In this sense, this algorithm still relies on deterministic methods to perform this task.) In the final stage, the algorithm then consults its "knowledge-based rules" to decide which of the matching methods was correct. These knowledge-based rules are never concretely defined in the paper, but it is stated that these rules are stored as the values for a dictionary in which known words are the keys; each word is subdivided into a list of its uses sorted by salience, and these uses then correspond to a list of contexts which that use is commonly found in. If a matching context is found, then that value of the word is selected. In order to have a better understanding of these contexts, each sentence is analyzed alongside with the sentences that immediately precede it and succeed it, in order to give the algorithm a broader perspective of the sentence in the center.

This algorithm most clearly resembles the pattern of human thought behind word segmentation, which we know to be correct most of the time. However, as promising as it may seem, the author fails to give concrete implementations of the structures they describe, never giving so much as pseudocode for this algorithm. As such, I question the viability of defining these "knowledge-based rules". Nevertheless, the idea of looking at three sentences at a time to have a better understanding of overall context seems to be sensible, and I believe that such a method of incorporating semantics into the parsing process would augment the morphological-syntactical method as implemented and would improve its accuracy.

3 Existing Tools Used

Unlike Chinese, Japanese words are subject to inflection and agglutinative grammar – thus, a functional word segmenter must be robust to these properties. Chinese, an extremely isolating language, is able to be segmented with only a dictionary because the grammar guarantees that each word will appear in its dictionary form. For Japanese, however, we must be able to recognize all possible forms of an inflected word. Thus, to accomplish this task, I have modified a Chinese dictionary matching model to run possible words through a Japanese transducer that will detect their original form (if there is one), and then check to see if that word is contained in a dictionary of the plain forms of Japanese words, rather than all possible inflections.

3.1 CWS Model

For the CWS Model, I used Wei En's statistical tokenizer built for Apertium as part of Google Code-In in 2013 as my base. En's paper is more of an analysis of the performance of his tokenizer rather than a description of its methodology, so the methodology that I describe here will be based on my

understanding of his code. Furthermore, little of En's original code is used in the new model, so I will spare the finer details of his method.

This program, in short, depending on the arguments given on the command line, can either return all possible segmentations, or be trained on a given sample text, and then return the learned segmentation. The segmentation methods available are Left-Right Longest Match, Right-Left Longest Match, and statistical, all of which were discussed in Section 2. It requires a dictionary file to be given as an argument so that it may check the validity of words by looking them up in this dictionary.

In the main file of the repository, tokenise.py, the first action in the main function is to parse the dictionary file given. The dictionary file is assumed to be of a specific format: one entry on each line, with each entry of the form [form]:[root form of word][<all>,<grammatical>,<tags>]. These tags are separated by comma, and detail every possible reading. This format is seen in the following example, taken from the dictionary included in En's repository:

不密:不密<adj>

Fig. 8: Example of dictionary entry
In En's implementation

This dictionary file becomes a Dictionary class object, as defined in the file dict.py. In the `__init__` function of the Dictionary object, each line of the dictionary is parsed and turned into an Entry object, which has three member variables, word, root form, and tags. So the above line, when parsed into an entry, would have the string "不密" as its `self.word`, "不密" as its `self.root_form`, and `["<adj>"]` as its `self.tags`. This is then added to the Dictionary object's tags member variable, which is a map from an Entry's `self.word` to a list of tuples containing its `self.root_form` and `self.tags`.

This is not all that the Dictionary class is comprised of. For each Entry object, it then separates its `self.word` into a list of the characters that are in that word, and then gathers information about each character's position in that word for the use of statistical parsing later on. To my understanding of this section, the program calculates each character's number of appearances at a certain "state", with this "state" meaning positional index in a word. For every state at which a character is found, it is then put into the `self.transitions` map, which maps tuples of the form (state, character) to the maximum number of states it can possibly have. After performing this for every character in an entry word, the program then adds the current state to `self.finals`, a list of integers, which keeps track of the number of states in a given word[1].

After it has created its Dictionary object, the function then uses this as the sole parameter for the creation of a Tokeniser object. This class, as described in tokeniser/tokeniser.py, needs only a dictionary file to function.

[1] My understanding of this section, and En's use of "states" throughout his

Next, the main() function then checks to see if the flag for "give all possible segmentations" has been provided by the user. If it has not, then it will train on the provided training file, a pre-segmented sample of text, and counts the frequency of every word in the file and then scores possible segmentations based on their frequencies, with a higher frequency earning a better score. If this flag has not been raised, then it will tokenize the text multiple times, each time using a different one of the three segmentation methods it knows.

It then separates the input file, or the file the user wants tokenized, by lines. Each line is then passed to the tokeniser object as a string of text to be tokenized. This tokenization happens in the tokenise method of the Tokeniser class. This method somehow utilizes the dictionary's knowledge of states and their relation to characters to return both possible segmentations of the text, LRLM and RLLM. These both give the maximum number of words possible, splitting two characters that could together make one word into two separate words (if valid) no matter what. All these segmentations are then returned as Unit objects. If the Tokeniser has been trained on a sample text, then it uses the rankings of words it has calculated to predict the most likely of all the possible segmentations, meaning that it would keep those two characters together as one word if that were more likely than each of those characters being separate words.

After this process, it then, for each line, prints out the tokens separated by spaces. After completing this process for every line in the document, it will have fully tokenized the text and outputted it to the user.

3.2 Japanese Morphological Transducer

For the morphological transducer, I used Swarthmore alumnus Darbus Oldham's transducer of Japanese. This transducer, written using Apertium tools, is capable of taking segmented Japanese word as input and returning each word along with its plain form and tags indicating what inflectional processes it underwent. This function of the transducer is used in its exact form (with a few feature extensions) in the new model, so I will give a detailed outline of the original code.

There are two main files that comprise Oldham's work; the .lexc file and the .twol file. The former describes the lexicon of Japanese, while the latter describes the phonological and orthographic transformations. The lexicon file, which will be detailed first, is made of three parts: categorical definitions, morphotactics, and lexicon.

```
%<n%>   ! Noun
%<pn%>  ! Proper noun
%<v%> ! Verb
%<adj%> ! Adjective
%<adv%> ! Adverb
```

```
%<cnjcoo%> ! Coordinating conjunction
%<ij%>  ! Interjection
%<prn%> ! Pronoun
%<cop%>    ! Copula
%<num%> ! Numeral
%<det%> ! Determiner
```

Fig. 9: Subsection of tag definitions
for parts of speech

The first section of the lexicon file defines part(s) of speech, subcategory, and grammatical tags. (The exclamation points indicate comments in the code, so anything written after them is ignored by the compiler and included for the reader's benefit only.) The code in Figure 9 is from lines 5-16 of the file, and defines the tags given to parts of speech. In the transduction process, every word must be assigned one of these parts of speech, by the nature of language: parts of speech define the possible categories of a word, and every word must belong to one of these categories, or else it cannot exist in the language. Oldham did not create an exhaustive list of parts of speech, but this was sufficient for creating a basic transducer of Japanese.

```
%<vol%>  ! Volitional
%<neg%>  ! Negative
%<past%> ! Past
%<imp%>  ! Imperative
%<npst%> ! Non-past
…
%<caus%>  ! Causative
%<pot%>   ! Potential
%<pass%>  ! Passive
```

Fig. 10: Sample of grammatical tags used
for verbs as defined in the .lexc file.

In contrast to the parts of speech tag, which must be assigned to all words, the tags in Figure 10 define verb suffixes, which attach to the end of a verb when it is inflected. Also unlike the POS tag, multiple grammatical tags can be assigned to a verb, whereas only one POS tag can be assigned to a word. For example, we will run the following word through the transducer:

食べ - させ - なかった
tabe-sase-nakatta
eat-CAUS-NEG.PAST.IFML
"[I] did not make [her] eat."

Fig. 9: Gloss of example Japanese text (one word).

We then see the following output:

```
^食べさせなかった/食べる<v><caus><ifml><past><neg>$
```

Fig. 11: Transducer's analysis of the word in Figure 9.

As we can see, the transducer was able to recognize that the part of speech of this word was a verb, as indicated by the verb tag. In addition, it was able to infer inflectional information about this verb, specifically, that it was in causative form, negative form, and past form. This is the correct analysis of this word, based on the gloss given above.

While some of these verb tags can occur in conjunction with each other, some must always appear in contrast. For example, a verb can only have one tense: therefore, we would hope our transducer never assigns a verb both the past and non-past tag at once. And indeed, it never does, as the code defines this rule not explicitly but rather passively, as a product of the morphotactic declaration.

Later on in the lexicon file, Oldham explicitly declares what suffixes are indicative of what forms. This section is aptly under the morphotactics header. In this part of the code, combinations of tags are paired with the combinations of characters that they correspond to. In the transducer's mind, the Japanese characters that comprise the words it parses are nothing more than meaningless symbols, and the definitions in the morphotactics section of the transducer tell it that, if it receives a certain ordered set of characters as input, to return a certain set of tags as output. Japanese has separate forms for its past and non-past forms that only occur in contrastive positions and are made of distinct character combinations. Therefore, the transducer never has to encode these rules as a heuristic: the morphology of the Japanese language itself, which inform the morphotactics of the transducer, has already done that for us.

The last key section of the lexicon file is the lexicon itself: this is the transducer's dictionary, the thing that gives it the knowledge to verify if a given combination of characters is a valid, or *known* word. Below is a sample of the lexicon encoding:

```
媒介:媒介 NounTag ; ! ばいかい  intermediary
空中:空中 NounTag ; ! くうちゅう air
…
静か:静か naAdjTag ; ! しずか  quiet
多い:多 iAdj ; ! おお  many
…
食べる:食べ RuVerbInflCP ; ! eat
落とす:落とす UVerbInflCP ; ! drop
```

Fig. 12: Samples of dictionary entries in the .lexc file.

As demonstrated in the code in Figure 4, a dictionary entry is not just the word itself. The word is provided in its plain form, or *jishokei*, which literally translates to "dictionary form", on the left of the colon. The item immediately to the right of the colon is the stem of the word, or its root form.

In his implementation, Oldham slightly deviates from the rules of Japanese morphology as given by linguistic analysis to simplify the transduction process: as described in the section giving an overview of Japanese linguistics, the true root of Japanese U-verbs end on a consonant (or occasionally a vowel), but because of the Japanese orthography, these true endings are always followed by an ending vowel that is determined by the inflection. In the word for "eat" in the above figure, 食べる (*ta-be-ru*), Oldham provides 食べ (*ta-be*) as the stem, dropping the る (*ru*), which truly is only a grammatical suffix not connected to the root of the word. But for the U-verb in the next line, 落とす (*o-to-su*), Oldham's code states that the stem is also the first two characters, or *oto-*, when in fact the true stem of this verb is *otos-*, which cannot be represented in the Japanese script, and the *-u* suffix indicates the informal non-past form of the verb. Again, this is a representational limitation of *kana*, not a fault of Oldham's analysis, but another option more in line with the linguistic analysis would be to encode the words in a different orthography, such as *romaji,* or use what is called the stem form of the noun in Japanese thought, the *i*-ending (ex. *o-to-si*) even though this is actually the gerundive form of the verb rather than the stem. Both of these proposed methods, however, would add more steps to the transduction process, and so it is up for debate whether it is better to follow the linguistic analysis or use tricks such as the ones Oldham used to simplify the .twol file's syntax.

The .twol file, on the other hand, defines the orthography of the target language, rather than the morphology. The first half of this file simply defines valid characters in this language. Again, to a transducer, the characters that form words are nothing but meaningless symbols to be identified, so this section of the code just serves to tell the program which ones it should identify. Japanese poses an interesting challenge in this task of providing the valid characters of a language: Japanese also uses *kanji*, of which there are about 2500, and the .twol file does not currently contain all possible characters, but this would nonetheless have to be done in order to make the transducer complete. In its existing form, Oldham only lists 100 of the most common *kanji* in the .twol file.

Also in the character list are the Arabic numerals, and the other two sets of characters, *hiragana* and *katakana*. As discussed in Section 1, the former is used mostly for commonly-appearing lexemes, ranging from grammatical particles to basic words. *Katakana* is mostly used to indicate that a word was borrowed from a different language. These character sets, referred to as *kana* collectively, are

comprised of 46 characters each, making for double that amount of characters in all.

The .twol file also categorizes these characters for the benefit of the transducer. For each one of the five vowel sounds in Japanese, Oldham groups together all the characters that end with that vowel sound. He also encodes the phonological transformations that verbs undergo when changing inflection as in the example below:

```
"{a} changes the end of the verb to Ca"
Cu:Ca <=> _ %>: %{a%}:0 ;
   where Cu in ( う く ぐ す む ぶ る )
         Ca in ( わ か が さ ま ば ら )
matched ;
```

Fig. 13: A rule describing the inflection of verbs in their dictionary form ending to their negative form ending.

This is as simple as changing the last vowel sound, but because both consonants and vowels are represented by a single character, the transducer must be careful to preserve the final consonant sound while changing the character to reflect the new vowel sound. The .twol file documents these rules by first recording the archiphonemes that will be used in the .lexc file to call for a change; Oldham then categorizes the vowel sounds together, and then defines these vowel transformations to the transducer in the form of rules as in the example above.

4 Proposed Tool

While En's model performs well on Chinese text, and is supposed by him to be able to work for any spaceless language, it places the bulk of its work on the dictionary file, which it expects to have every possible grammatical permutation of a word. Japanese, however, uses a grammar that follows a strict pattern of attaching certain inflectional and syntactical endings to the stem of a word, and calculating every possible permutation would be akin to lemmatizing every word form in the language, which, although basely finite, is computationally time-consuming as well as redundant in terms of work done. So, instead of constructing every possible permutation in advance, the model I propose works by instead deconstructing every word it is given to find its root form and grammatical tags. This ensures that we are only analyzing the words we need to, the fewest number possible, rather than every combination imaginable, the maximum.

Thus, my model eschews the dictionary file and Dictionary class altogether. Instead of referencing a dictionary to check the validity of words, this model instead runs them through the transducer, and uses the results of that to inform its judgment of accuracy. Removing the Dictionary object also removes

the tokenizer's capacity to store states and score tokenizations based on a training file, since the information about states was stored in the Dictionary object. However, states cannot be calculated in this transducer-based model anyway, as En's version relies on knowing all possible words in advance, and using that information to gather data about states before the tokenization process occurs. With the unweighted transducer model, we know nothing about the language in advance of checking a proposed token's validity, which occurs during the tokenization process. So, one weakness of this new model is the fact that it loses its information about states, but given that states were poorly documented, that information probably would not have benefitted this model much anyway.

As alluded to in the prior paragraph, this model checks the validity of *all* possible combinations of characters in the given text. To do this, we use dynamic programming to ensure that we calculate all possible combinations while also never doing redundant work. It is also reduced entirely into one class, tokenise.py, and no longer needs classes such as Dictionary and Trainer. The only dependencies outside the script and the libraries it is built on are an input file, and an HFST file, which are both passed in as arguments on the command line.

The bulk of the code resembles En's CWS, as that code was used as the basis for this code. The argument parsing and input-file processing remains more or less the same. The difference comes in the Tokeniser class's method tokenise() – the code in this function is entirely new.

Before looking at the tokenization process itself, it would be beneficial to first look at the new method of verifying a word's validity. En's model did this with the dictionary, but this model does it with the transducer. This validation process happens in Tokeniser.isWord(), a new method for this class.

```
def isWord(self, ls):
      string = ''.join(ls)
      input_stream = hfst.HfstInputStream(self.hfst)
      analyser = input_stream.read()
      fullout = analyser.lookup(string)
      if len(fullout) == 0:
            return False
      output = fullout[0][0]
      items = output.split("<",1)
      token = Unit(items[0], items[1])
      return token
```

Fig. 13: The function `isWord()` in the new tokenizer that checks to see if a given string comprises a valid word.

This function utilizes the HFST Python library, which allows interaction with HFST-format binary transducers in the Python interface. If the HFST analyzer did not find the word to be valid, it will return the empty set, and the isWord() function will return the Boolean value False. If it does find valid results, it will return them in the form of a tuple, with the first item being the list of all parsed words and their tags, and the second item being the error code (which we hope would be 0). Since we are only ever looking for one word at a time, we take the first item in the list of words and tags found. We then separate that into the word itself and its tags, which are stored in the Unit cohort. So, in this "not fail" case in which we have found a valid word, we then return that word and its tags encoded in a Unit cohort.

The isWord() function serves as our method of checking the validity of a certain combination of characters. To compute this combination of characters, the tokenizer utilizes dynamic programming to reuse work. The basic premise of dynamic programming is that we have an array that tells us the answer to our previously-solved sub-problems (Kleinberg and Tardos 2006, p. 251). We will call this array D, and it will be the length of the number of characters in our string, since we define sub-problems as the string from the first character to any given character. The initialization of this array occurs in the following code:

```
chars = list(text)
n = len(chars)+1

D = np.zeros([(n),], dtype=object)

for i in range(0,n):
      D[i] = "FAIL"

first = self.isWord(chars[0:1])
if first != False:
      D[1] = [first]
```

Fig. 14: The initialization step of the dynamic programming process used in this implementation.

We initialize all items in the array to FAIL with the assumption that the array will check to see if every sub-problem passes, and update the values of the ones that do. We leave the first item in the array, D[0], to FAIL, because this represents the valid words found in the empty string, which, of course, is none. Our base case is D[1], which looks for the valid words in the string that is just the first character of the input string. This will either leave the value of D[1] at FAIL, or, in the case that the first character is a valid word by itself, update D[1] to be a list containing just that word as a Unit. With this base case set, we now are able to iterate over the rest of the string to find the number of valid strings within the entire input string.

```
for i in range(2,n):
      for j in range(0,i+1):
            right = self.isWord(chars[i-j:i])
            if right != False:
                  left = D[i-j]
                  if left != "FAIL":
                        D[i] = left + [right]
                  else:
                        D[i] = [right]

tokens = D[-1]
```

Fig. 15: The loop that performs the inductive operation on the entire string.

The outer for-loop serves to define the last character of the substring we observe. Since we already know if the first character by itself is a word, this for-loop starts by looking at the first character and the second character (if there is one). This for-loop runs for the number of characters, so for each iteration, the next character is added to the substring.

The inner for-loop serves to define the point at which we break our current substring into two halves, left and right. The variable j ranges from 0 to i, and then is used in defining the breaking point: this means that at the first iteration of j, the right half will be empty and the left half will be the entire substring, and for each iteration, the right half will take the last character of the left half, so that by the last iteration, the left half will be empty and the right half will be the entire substring. For each right half we have, we check to see if that right half is a valid word: if it is, we check D's entry for the substring that is the left half. That look-up operation will either return FAIL, in which case, we set D[i] to be only the right substring as a valid word in the substring up to character i; or, the left half can be validly tokenized, in which case we union the list of tokenized words from the left half with our single-item list of our tokenized word from the right half and set that as the value of D at that index.

Through this process, we increase the length of the substring one at a time until, at the end of the outer for-loop, we have looked over the entire string. So we can return the value of D for the last character index in the input string to get all the tokenized words in that substring, if any. This process is illustrated in the following figures, using the example string "abcdefg":

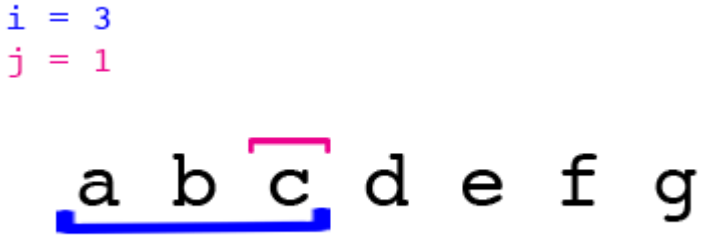

Fig. 16: An example of the two for-loops' substringing process.
In this example, the counter for the outer for-loop, variable *i*,
is 3,and the counter for the inner for-loop *j* is 1.

The string of characters partitioned by the blue bracket here indicates the substring selected by the outer for-loop. This substring always starts with the first character, and increments its length by 1 as the variable *i* is incremented. The string of characters partitioned by the pink bracket represents the substring referred to by the variable "right" – this is the right half of the substring which the outer for-loop is currently looking at. As such, the left half would then be the part of the blue substring not contained within the pink substring. The dynamic programming function then checks to see if "right" is a valid word, using the isWord() function; if it is, it will then use the dynamic array D to see if the left substring is also a word, or a combination of valid words. If both halves are words, it will save all these valid words as the value for D[i]. If at least one half is does not contain a valid word, then the value of D[i] is then FAIL. It then increments *j* (in this case, that means that *j* = 2 gives the right substring "bc") and then repeats this process of defining the value of D[i] until the pink substring is the same as the blue substring. It then increments *i* and repeats this process starting from *j* = 0 once more. By this process, it checks all possible left/right segmentations of all possible substrings of length 0 to *n*, where *n* is any integer less than or equal to the length of the entire string.

This list of Units that were found, if it exists, is then returned to the main function that then prints out each Unit's word followed by its tags followed by a space. Oldham's transducer parses grammatical particles as grammatical markers rather than words of their own, so particles that were in the input text will not appear in the output text, and will instead manifest in the tags of the words they are suffixed to. As stated in the introduction, the role of particles in Japanese is still debated, but any changes to the tokenizer's interpretation of them would have to be made in the transducer rather than the tokenizer itself.

5 Test Results

The basic tests of this tokenizer will be to see how it deals with three different types of input: strings containing only valid words (i.e. words in the transducer), strings containing some valid words and some invalid words, and strings containing no valid words.

The first test is on the following Japanese sentence, which is completely valid in Japanese, and comprised of only words that are already in the transducer:

母親が　　子供に　　野菜を　　食べさせなかった。

hahaoya-ga *kodomo-ni* *yasai-wo* *tabe-sase-nakatta*
mother-NOM child-DAT vegetable-ACC eat-CAUS-NEG.PAST.IFML
"The mother did not make the child eat the vegetable(s)."

Fig. 17: Gloss of first test sentence.

As stated in Section 2, Japanese does not express the number of nouns, so it is unclear in this sentence whether the word for "vegetable" is plural or singular. When we include this sentence in the input file, the output is the following:

```
母親<n><sg><nom> 子供<n><sg><dat> 野菜<n><sg><acc> 食べる
<v><caus><ifml><past><neg> 。
```

Fig. 18: Transducer's analysis of the sentence in Figure 16.

The output for this file is almost identical to the gloss given in Figure 11, with every tag being preserved. The transducer assumed that the number of every given noun was singular, which, for both "mother" and "child", is accurate. For the unclear case, "vegetable", it assumed singular as well, and while not necessarily inaccurate, the semantic number of this noun depends on the context of the sentence, which the transducer cannot infer. Or, in truth, it is rather the case that the Japanese language deems the morphological number of "vegetable" unnecessary in this sentence, and so it is left unclear, but the transducer, which designed to assign explicit case, defaults to singular. Perhaps a better tag for the transducer to give a noun without explicit plural marking would be something like "number unmarked" to signify that the number is not declared.

This test case was also interesting because, at this point, for the characters 母 and 親, I had included only the following two words in the transducer's lexicon:

```
母:母 NounTag ; ! mom
母親:母親 NounTag ; ! mother
```

Fig. 19: Valid character combinations for the two characters 母 and 親 as defined by the .lexc file at runtime.

The transducer, in its dynamic programming process, saw 母 as a valid word, kept that as the entry for that character alone, and then, on the next iteration, saw that 母親 was a valid word, but 親 alone was not, and so chose the option that produced no failing output, which is the two characters grouped together as one word rather than one single-character valid word and one failing single-character. However, 親 alone is indeed a valid word in Japanese, meaning "parent". The next test will be to see how the transducer behaves when

presented with multiple tokenization options for a given combination of characters.

There is no need to give the output for this new test, because the output is exactly the same as the output in Figure 18. This was actually contrary to my expectation of the program: the dynamic programming model I based my code off returned the maximum possible number of words, and so I expected the tokenizer to parse 母 and 親 as two separate words, with the former being tagged as absolutive and the latter, which is directly attached to the particle, being tagged as nominative. However, this program instead prefers the longest possible combination of characters of the largest possible number of words, and so the two-character word is chosen, which is actually the correct choice. Although, one could claim that the two-character word 母親 is actually a compound of the two separate words 母 and 親, so the argument could be made that the two-word parsing is also valid. However, the intuition of a native speaker[2] is that it is one word. So, in a surprising twist, this model does not immediately assume that the segmentation that gives the maximum number of words is the correct one, a mistake that LRLM and RLLM models make.

This peculiarity is due to a lack of a check statement within the original code. Specifically, when the program finds a valid segmentation for both the left and right halves of the substring, it immediately assigns that segmentation to the DP array. It never considers the case in which the DP array already has a valid segmentation for that substring. As such, it inherently prefers the segmentation found later on, or in other words, the substring found at a larger value of *j*. This means that it prefers the segmentation in which the right half of the substring is the largest possible single word, which in turn means it prefers the segmentation with the smaller number of total words within it. So by not explicitly declaring that the fewest word segmentation should be preferred, the program does, in fact, prefer it.

The next test will include some words that the transducer recognizes as valid, and some that it does not. The test input will be:

おまえは	もう	死んで	いる。
omae-wa	*mou*	*shinde*	*iru*
you-NOM	already	dead-GER	exist.AN.NPST.IFML

"You are already dead."

Fig. 20: Gloss of second test sentence.

The first two words, おまえ and もう, have been entered into the lexicon and thus are valid words. The last two words are actually thought of in Japanese as one word, the resultative conjugation of 死ぬ, "die". The resultative is formed by inflecting the verb to its gerundive form and then affixing the existential verb for

[2] Thank you Yosuke Higashi '18

animate objects, いる, in its plain form as an auxiliary verb. This is a valid word in Japanese, but the transducer is not yet capable of detecting the gerundive form of verbs, and いる is not included in the lexicon, so both parts of this word will be rejected by the transducer.

As expected, the tokenizer's response to this input is simply to print out its error message, the string "No complete tokenization found". However, when the tokenizer is run in debug mode, we see that at the twelfth iteration of the outer loop and the fourth iteration of the inner loop, where the left substring is "おまえはもう" and the right "死んでいる。", that while the tokens for the left substring are indeed ["おまえ", "もう"], the tokenizer is not able to find a valid segmentation of the right string, and so, since it is not able to validly tokenize the entire string, it returns FAIL for this string. If the transducer were updated to be able to handle the resultative form of verbs, the tokenizer would then almost certainly give the correct tokenization of this sentence. What this demonstrates is that if the tokenizer is not able to find any split of the string (and its substrings) in which both the left half and the right half have valid tokenizations, it will return FAIL for the entire string.

This example sentence with an invalid word already gives us the answer for how the tokenizer would handle completely invalid input. Due to the nature of dynamic programming, every possible sub-problem is solved, and one of those sub-problems, 死んでいる, was made of entirely invalid input, and the tokenizer failed to parse this. So, we see that if any substring of the input string is invalid, including the case where the substring is the entire string, the tokenizer will fail and be unable to tokenize any of the words in the string, even the valid ones.

6 Conclusion and Future Work

This new model of the tokenizer accomplished two main tasks: correctly tokenizing valid input phrases, and returning the grammatical tags of the tokens it found. The latter distinguishes this model as not just a tokenizer, but potentially also as the first step in a complete text-parsing process. These grammatical tags could be used to understand each word's role in the sentence and thus better inform the algorithm's analysis of the meaning of the phrase, which would be a remarkable accomplishment for future work.

However, this model is not without its weaknesses. Most striking is the fact that it can only tokenize words that are in the transducer's lexicon; any other word in the input string, even if it is a valid word, is rejected by the tokenizer, which causes the entire input string to be rejected. This could be amended by two different methods, or preferably both: completion of the transducer's lexicon, or modification of the tokenizer to ignore bad tokens and still return all valid tokens.

The former is a work-intensive process, but does not demand too much brainpower. This could be done, for example, by creating a separate script that parses the words providing by Universal Dependencies' Japanese corpus. This corpus contains a vast number of example sentences, each one pre-segmented, and with the inflectional information for each word in the sentence given. All that this script would have to do is, for every word given, find its root form in the analysis attached to it, and add that word to the lexicon along with its part of speech information, if the lexicon does not already contain it. Such a script would massively expand the capabilities of this tokenizer.

The latter method requires a conceptual understanding of the dynamic programming process, which is unfortunately far from trivial. The tokenise() function would have to be modified to, in the event of the one half contains a fail, still keep the other half of the substring, and give the array at that index a list containing those valid words and the combination of characters that are *not* a word, so that it is accurately represented as a bad token in the output. The challenge of this is making sure that we maximize the number of good tokens found, as we know for sure that for each iteration of the outer loop, the inner loop will return a variety of possible word lists, whereas currently we can rely on it providing either one valid one or none at all. But while difficult, this would give the tokenizer the ability to handle unknown tokens, which is computationally a simpler task than providing every single valid word to the lexicon.

These two approaches in conjunction would give the tokenizer the ability to accurately tokenize and tag a large amount of words, and robustly handle the (hopefully) few words it does not know. Additionally, some of the heuristics discussed in Section 2, such as word order denoting grammatical role, are not yet utilized in the current tokenizer. Another possible extension would be using assumptions made from word order to tag words that the transducer cannot accurately tag: this would either be in cases of words not in the lexicon, or of words not marked with particles, as often occurs in casual speech.

Another weakness of the current transducer is its limited morphological definitions. It still does not yet have all possible verb conjugations defined, but in fairness, some of these cases would be hard to define. For example, the passive auxiliary morpheme for *Ru*-verbs, *rare*, is also the potential auxiliary morpheme for *Ru*-verbs. The transducer currently does not define the potential state, and instead assumes every instance of this auxiliary morpheme to be indicative of the passive voice. If it were to define both, it would either need a way to tell, based on context or other hints from the text, which of these two cases the instance represents, or store both possible interpretations and return both to the user, and leave it up to the user to decide which is correct.

Also, as alluded to in the methodology section, the transducer currently does not show the original particles in its output and instead manifests them in the tags of the words they were suffixed to. Whether or not particles are themselves words is still debated in the linguistics community (Lang, 2016), and whether or not maintaining particles is necessary or not is dependent upon the task itself. If this tokenizer is being used as the first step in, say, an English

translator, then the tags alone will be sufficient. But if the goal is simply to insert spaces where applicable into the original text, then the output should still possess every single character that was in the original text. But to change this functionality, one would need to change the transducer, whose implementation was not modified at all for the sake of this tokenizer.

Also, as alluded to in the previous section, I was surprised by the fact that despite never being explicitly told to prefer segmentations with less words, the tokenizer does so anyway. However, in its creators opinion, this bias is actually the correct one for the program to make: given the nature of Japanese, particularly *kanji*, words placed next to each other without any sort of syntactic delimiters such as particles between them tend to compound (Kobayashi, 1994). And for the tokenizer to recognize such a compound word, then it must have been included in the transducer's lexicon. So the question of whether or not two words can validly compound or not is one the transducer makes, not the tokenizer, and so the tokenizer is making accurate decisions according to the transducer's lexicon.

Nevertheless, in its current form, this tokenizer, in its creator's opinion, rivals both the statistical tokenization model because it uses morphological information to identify possible tokens, and the linguistic model, since it considers all possibilities from the start instead of limiting the scope of its hypothesis space based on heuristic assumptions. One last thing I wish I could have done is compare its performance to that of existing Japanese tokenizers.

7 Works Read

Huang, Chu-Ren, and Nianwen Xue. "Words without Boundaries: Computational Approaches to Chinese Word Segmentation." *Language and Linguistics Compass,* 2012, Blackwell Publishing Ltd.

Kleinberg, Jon, and Éva Tardos. *Algorithm Design*. Pearson Education, Inc., 2006.

Kobayashi, Yosiyuki, et al. "Analyis of Japanese Compound Nouns using Collocational Information." 25 Dec. 1994, Tokyo Institute of Technology.

Kubota Ando, Rie, and Lillian Lee. "Mostly-unsupervised statistical segmentation of Japanese kanji sequences." *Natural Language Engineering,* 3 May 2003, Cambridge University Press.

Lang, CJV. "Thoughts on the Universal Dependencies proposal for Japanese." Web blog post. Cjvlang.com. Spicks & Specks, September 18th, 2016. Web. December 10th, 2017.

Li, Wei. "On Chinese parsing without using a separate word segmenter." *Communication of COLIPS,* 25 Sept. 2000, Hong Kong University of Science and Technology.

Li, Wei. "The Morpho-Syntactic Interface in a Chinese Phrase Structure Grammar." Jan. 2001, Simon Fraser University.

Miyamoto, Edson T. "Case Markers as Clause Boundary Inducers in Japanese." *Journal of Psycholinguistic Research,* Jul. 2002, Plenum Publishing Corporation.

Shibatani, Masayoshi. *The Languages of Japan*. Cambridge University Press, 1990.

Wu, Zhijie. "A Cognitive Model of Chinese Word Segmentation for Machine Translation." *Meta LVI,* 2011, Nanjing University of Science and Technology.

Yao, Yuan, and Kim Ten Lua. "Splitting-Merging model of Chinese word tokenization and segmentation." *Natural Language Engineering,* 3 Mar. 1998, Cambridge University Press.

8 Acknowledgments

First, I would like to thank my thesis advisor, Jonathan North Washington, for aiding me and being willing to deal with my faults all semester. I would also like to thank Donna Jo Napoli for helping me out along the way. I also owe credit to Bryce Wiedenbeck and Lila Fontes, who both taught Algorithms this semester. Bryce was my professor, and he was the one to first teach me about dynamic programming, which I knew on the spot would be relevant to this work, and Lila was the one to help me when I ran into trouble during the implementation, and even go a step beyond that by showing me a more efficient approach. I would like to thank all the professors I've had in the CS, Linguistics, and Japanese Departments here at Swarthmore for giving me the foundation I needed to do this work.

To my student readers, Ziting Shen and Elsher, I congratulate you on finishing your theses and thank you for the support throughout the semester. I would also like to thank Yosuke Higashi for being a native Japanese speaker who was in the room when I needed a native speaker's opinion, as well as Madeleine Pattis and Christine Lee for bringing me cookies while I was writing this.