2015-01-01

Japanese Computational Lexicon: A Computational Dictionary Of Japanese Verb Forms

Mayumi Kobayashi
University of Texas at El Paso, mayumi.kobayashi930@gmail.com

JAPANESE COMPUTATIONAL LEXICON:

A COMPUTATIONAL DICTIONARY OF

JAPANESE VERB FORMS

MAYUMI KOBAYASHI

Department of Languages and Linguistics

APPROVED:

Nicholas Sobin, Ph.D., Chair

Jon Amastae, Ph.D.

John McClure, Ph.D.

Charles Ambler, Ph. D.
Dean of the Graduate School

JAPANESE COMPUTATIONAL LEXICON:

A COMPUTATIONAL DICTIONARY OF

JAPANESE VERB FORMS

By

MAYUMI KOBAYASHI

THESIS

Presented to the Faculty of the Grauate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

Department of Languages and Linguistics

THE UNIVERSITY OF TEXAS AT EL PASO

December 2015

ABSTRACT

This paper presents two main topics: a linguistic analysis of Japanese morphological lexicon focused on verb conjugation and a computational analysis of Japanese verb conjugation in Prolog programming. Prolog is one of the computer programming languages and is used to solve a problem logically with a number of statements called 'clauses,' which are defining various relationships of the entities. With respect to the linguistic aspect, this paper discusses Japanese verb formation, its conjugation patterns, twelve basic forms of Japanese verbs, and a Japanese morphophonemic process called "音便 (*onbin*)" by investigating 304 regular verbs and 2 irregular verbs. Among the several different conjugation patterns defined by linguists, this paper focuses on the well-known model by non-native Japanese learners called the 'morphological approach.' The primary goal of this research is to develop an electronic Japanese verb conjugation dictionary using Prolog programming. I apply all analyzed traits of Japanese verb conjugation to the Prolog programming to process 306 verbs to elicit 23 form variations of consonant verbs and two irregular verbs and 24 form variations of vowel verbs. This electronic Japanese verb conjugation dictionary can help non-native Japanese learners to learn fundamental Japanese verb formation and its conjugation with various forms.

Keywords: Japanese verb conjugation, Morphological conjugation model, Morphophonemic process, 音便 (*Onbin*), Prolog

TABLE OF CONTENTS

Page

ABSTRACT……..…….iii

TABLE OF CONTENTS……..… iv

LIST OF TABLES……...…vii

LIST OF TREES……..viii

LIST OF OUTPUTS FROM THE PROLOG PROGRAM……..… ix

CHAPTER

1. INTRODUCTION……..…….1

1.1 Background of Natural Language Processing……....1

1.2 Japanese Verb Conjugation Patterns……...…1

1.2.1 Traditional Japanese Verb Conjugation Model……...…2

1.2.2 A Morphological Verb Conjugation Model……...…4

2. LANGUAGE STRUCTURE ANALYSIS…….6

2.1 Word Formation Rule Analysis……....6

2.2 Japanese Verb Formation Analysis……...…8

2.3 Word Formation Rule of Japanese Verbs……...…11

3. JAPANESE VERB CONJUGATION ANALYSIS…….13

3.1 Verb Forms……..…13

3.1.1 Imperfective Form/未然形(mizen kei)……13

3.1.2 Continuative Form/連用形(renyō kei)……14

3.1.3 Past Tense Form/過去形(kako kei)…….14

3.1.4 Polite Form/丁寧形(teinei kei)……14

3.1.5 Predicative Form/終止形(shūshi kei)……...…14

3.1.6 Causative Form/使役形(shieki kei) and Passive Forms/受身形 (ukemi kei)……...15

3.1.7 Potential Form/可能形(kanou kei)……...15
3.1.8 Imperative Form/命令形(meirei kei)……...15
3.1.9 Conditional Form/仮定形(kateikei) and Volitional Form/意向形 (ikou kei)……..…16
3.2 Verb Conjugation Groups……...16
4. MORPHOPHONEMIC PROCESS: *ONBIN*……....20
4.1 *Onbin* Process 1……..…20
4.2 *Onbin* Process 2…….….23
4.3 *Onbin* Process 3…….…….23
5. AUXILIARY SUFFIXES IN VERB FORMATION……...…25
6. PROLOG PROGRAMMING……...…28
6.1 What Is Prolog?.....28
6.2 Natural Language Analysis for Prolog Programming……28
6.3 Prolog Program in General Example……..…29
6.4 Japanese Verb Conjugation in Prolog…….32
6.5 Consonant Verbs and *Onbin* Process in Prolog……..…40
6.6 Vowel Verbs and Two Irregular Verbs in Prolog……...…44
6.7 The Process of Eliciting Japanese Verb Conjugation in a Proof Tree……46
7. DISCUSSION OF THE OUTPUTS IN JAPANESE VERB CONJUGATION PROLOG PROGRAM……..49
8. CONCLUSIONS AND FUTURE WORK……...58
REFERENCES……....60
APPENDIX 1: Final Prolog Program of Japanese Verb Conjugation…….…61
APPENDIX 2: Outputs of the Consonant Verb *"kaku"*……..…87
APPENDIX 3: Outputs of the Vowel Verb *"taberu"*…….90
APPENDIX 4: Outputs of the Irregular Verb *"kuru"*…….93

APPENDIX 5: Outputs of the Irregular Verb *"suru"*……..…96

CURRICULUM VITA……...….99

LIST OF TABLES

Table 1 "Five Grade Conjugation" Verb Paradigm of *"kaku"*……2

Table 2 "Upper One-tier Conjugation" Verb Paradigm of *"okiru"*……3

Table 3 "Lower One-tier Conjugation" Verb Paradigm of *"taberu"*……...3

Table 4 "ka-row irregular conjugation" Verb Paradigm of *"kuru"*……4

Table 5 "sa-row irregular conjugation" Verb Paradigm of *"suru"*……..4

Table 6 Verb Paradigm of the Consonant Verb "*kaku"*……17

Table 7 Verb Paradigm of the Vowel Verb "*taberu"*……18

Table 8 Verb Paradigm of the "*kuru*" Irregular Verb……18

Table 9 Verb Paradigm of the "*suru*" Irregular Verb……19

Table 10 *"Onbin"* Process 1 for Consonant Verbs Ending Root in *-t, -r, -u(w)*……..21

Table 11 Verb Paradigm for the Consonant Verb *"katsu"*……22

Table 12 *"Onbin"* Process 2 for Consonant Verbs Ending Root in *-n, -m, -b*…….23

Table 13 *"Onbin"* Process 3 for Consonant Verbs Ending Root in *-k, -g*……24

Table 14 The Sequence of Auxiliary Suffixes……25

LIST OF TREES

Tree 1 Tree Structure of English Verb Formation……7

Tree 2 Tree Structure of the Verb "walked"……..…7

Tree 3 Derivational Tree Structure of the noun "predictability" from the verb "predict"…8

Tree 4 Derivational Tree Structure of the noun *"oyogi"* from the verb *"oyogu"*……....9

Tree 5 Derivational Tree Structure of the noun *"awase"* from the verb *"au"*……..10

Tree 6 Derivational Tree Structure of the noun *"tabe-mono"*……...10

Tree 7 Tree Structure of Japanese Verb Formation……11

Tree 8 Tree Structure of Polite Past Form of the Verb *"kaku"*……..12

Tree 9 Tree Structure of Formal Past Negative Potential Conditional Form of the Verb *"taberu"*…….26

Tree 10 Word Formation Rule of Japanese Verb Conjugation……..…33

Tree 11 Proof Tree Structure of Formal Past Negative Form of the Verb *"kaku"*……..…47

LIST OF OUTPUTS FROM THE PROLOG PROGRAM

Output 1 Partial Output of the First Verb "*agaru*" and the Second Verb *"ageru"*……49

Output 2 The Feature Information of the Verb Form of *"kak-a-na-i"*……...52

Output 3 The Base Verb Stem of *"kaku"* in Each Form……....53

Output 4 The Verb Root of *"katsu"*……...…54

Output 5 The Formal Past Negative Conditional Form of Auxiliary Suffixes in Order….55

Output 6 The Verb Form of *"oyogu"* in the Informal Present Continuative Form……56

Output 7 Translation of the Verb *"aku"* in English and "wait" in Japanese……..…56

1. INTRODUCTION

1.1 Background of Natural Language Processing

How human beings acquire and produce languages is a major concern among linguists. Noam Chomsky argued that "children must innately be equipped with a plan common to the grammars of all languages, a Universal Grammar" (Pinker, 1994:22). That tells us that human beings are born with an innate ability to develop language competence as new born babies learn how to drink milk, craw, stand, and walk without any instructions. In the process of how human beings acquire languages, they find the common characteristics that languages share as principles and also encounter diverged variations from the commonalities as parameters. That is, most of the languages have a grammatical category, "a verb," but verb conjugation appears different in different languages. English and Japanese have verbs but the verbs of each language conjugate differently.

From the perspective of computer science, the research focuses on how this human natural language processing can be computationally executed. That is the fundamental idea of Natural Language Processing (NLP). Matthews (1998:3) addresses the computational approach to Natural Language Processing as follows:

> Since a large part of human communication is conducted effortlessly through natural languages such as English, Japanese, or Swahili, the ability of computers also to be able to converse in such languages will be one of the crucial components in making them 'human literate.'

With respect to the computational linguistic perspective, I investigate how Japanese verb morphology might be computationally executed and present a computational analysis of basic Japanese verb conjugation based on a programming language Prolog. With the analysis from the computational linguistic perspective, this Prolog program can be used as a learning tool for non-native Japanese learners.

1.2 Japanese Verb Conjugation Patterns

There are several different theoretical models offered by various linguists for Japanese verb conjugation. Two major Japanese conjugation models are introduced in this section. One is the traditional Japanese verb conjugation based on the Japanese syllabary *'hiragana'* that is taught to native Japanese speakers, and the other is the morphological conjugation model written with the Roman alphabet and used to introduce Japanese to non-native learners. In this paper, the morphological model is used because it is more useful

to non-native learners and for Prolog programming which only recognizes alphabetical letters. But before directly moving to the morphological model, this section discusses what traditional Japanese verb conjugation model is and how the traditional model integrates into the morphological conjugation model.

1.2.1 Traditional Japanese Verb Conjugation Model

This traditional verb conjugation model is usually introduced to native Japanese children to learn its conjugation pattern at school. The traditional model consists of three regular verb conjugation patterns and two irregular verb patterns based on how they are written in the Japanese syllabary characters, 平仮名(hiragana). The three main regular conjugation groups are "Five Grade Conjugation" called 五段活用(godan katsuyou), "Upper One-tier Conjugation" called 上一段活用(kami ichidan katsuyou), and "Lower One-tier Conjugation" called 下一段活用(shimo ichidan katsuyou). And there are only two irregular verbs: くる *"kuru"* meaning "to come" defined as "ka-row irregular conjugation" called カ行変格活用(ka-gyou henkaku katsuyou) and する *"suru"* meaning "to do" defined as "sa-row irregular conjugation" called サ行変格活用(sa-gyou henkaku katsuyou).

This traditional conjugation grouping is based on the conjugated patterns of each verb group. First of all, most verbs would be broken down into two portions, a base verb to be conjugated and auxiliary suffixes. Table 1 shows the verb paradigm of "Five Grade Conjugation" using the verb かく *"kaku"* meaning "to write." The verb paradigm includes the five basic forms; the imperative form as negation, the polite form as formality, the predicative form as a dictionary form, the conditional form as if-condition, and the volitional form as volition.

[Table 1] "Five Grade Conjugation" Verb Paradigm of *"kaku"*

	Base verb	Auxiliary suffix
<Imperfective form>	か***か*** (kak***a***)	ない (-nai)
<Polite form>	か***き*** (kak***i***)	ます (-masu)
<Predicative form>	か***く*** (kak***u***)	。(-[])
<Conditional form>	か***け*** (kak***e***)	ば (-ba)
<Volitional form>	か***こ*** (kak***o***)	お(う) (-o)

In the above table, you will notice that all forms of the base verb of *'kaku'* end with Japanese syllabary letters "ひらがな hiragana" ***か***(k*a*), ***き***(k*i*), ***く***(k*u*), ***け***(k*e*), ***こ***(k*o*). As in Roman

letter writing you will notice that the Roman spelling of these Japanese syllabary letters carry all five vowels ***a, i, u, e, o***; therefore, it is called "Five Grade Conjugation" in the traditional model.

However, "Upper One-tier Conjugation" and "Lower One-tier Conjugation" share the pattern in which each conjugated verb base ends with the same Japanese syllabary letter. Tables 2 and 3 show the verb conjugation patterns of the verb おきる *"okiru"* meaning "to wake" for "Upper One-tier Conjugation" and the verb たべる *"taberu"* meaning "to eat" for "Lower One-tier Conjugation."

[Table 2] "Upper One-tier Conjugation" Verb Paradigm of *"okiru"*

	Base verb	Auxiliary suffix
<Imperfective form>	おき (ok***i***)	ない (-nai)
<Polite form>	おき (ok***i***)	ます (-masu)
<Predicative form>	おきる (ok***i***ru)	。(-[])
<Conditional form>	おきれ (ok***i***re)	ば (-ba)
<Volitional form>	おきよ (ok***i***yo)	お(う) (-o)

[Table 3] "Lower One-tier Conjugation" Verb Paradigm of *"taberu*"

	Base verb	Auxiliary suffix
<Imperfective form>	たべ (tab***e***)	ない (-nai)
<Polite form>	たべ (tab***e***)	ます (-masu)
<Predicative form>	たべる (tab***e***ru)	。(-[])
<Conditional form>	たべれ (tab***e***re)	ば (-ba)
<Volitional form>	たべよ (tab***e***yo)	お(う) (-o)

As in Table 2 and 3, each "Upper One-tier Conjugation" verb and "Lower One-tier Conjugation" verb contains the same Japanese syllabary letter to conjugate such as き(ki) for "Upper One-tier Conjugation" verb ending and べ (be) for "Lower One-tier Conjugation" verb ending, respectively. The naming of the two models derives from these common letters in each conjugation model. When written, Japanese vowel alphabetical order あ(a), い(i), う(u), え(e), お(o) is set up vertically. In this order, the Roman vowel ***'i'*** in き(k***i***) is placed higher than ***'u,'*** so this is known as "Upper One-tier Conjugation;" and in "Lower One-tier Conjugation" the Roman vowel ***'e'*** in べ(b***e***) is placed lower than ***'u.'***

On the other hand, the two irregular verbs *"kuru"* and *"suru"* behave differently from these two regular conjugation verbs in their conjugation patterns. You can find their irregular conjugation patterns in Table 4 below showing the verb paradigm of くる*"kuru"* for "ka-row irregular conjugation" and Table 5 showing the one of する*"suru"* for "sa-row irregular conjugation."

[Table 4] "ka-row irregular conjugation" Verb Paradigm of *"kuru"*

	Base verb	Auxiliary suffix
<Imperfective form>	こ (ko)	ない (-nai)
<Polite form>	き (ki)	ます (-masu)
<Predicative form>	くる (kuru)	。(-[])
<Conditional form>	くれ (kure)	ば (-ba)
<Volitional form>	こよ (kayo)	お(う) (-o)

[Table 5] "sa-row irregular conjugation" Verb Paradigm of *"suru"*

	Base verb	Auxiliary suffix
<Imperfective form>	し (shi)	ない (-nai)
<Polite form>	し (shi)	ます (-masu)
<Predicative form>	する (suru)	。(-[])
<Conditional form>	すれ (sure)	ば (-ba)
<Volitional form>	しよ (shiyo)	お(う) (-o)

As you can see in the above tables, these two verbs conjugate irregularly compared with other regular verb conjugation patterns.

Thus, the traditional verb conjugation model consists of three regular conjugation patterns and two irregular conjugation patterns. A morphological approach to verb conjugation; however, is able to integrate these three regular conjugation patterns in the traditional model into two conjugation patterns by breaking down each Japanese syllabary letter into phonemes with using Roman letters.

1.2.2 A Morphological Verb Conjugation Model

The morphological verb conjugation model that is introduced to non-native Japanese learners in most Japanese grammar textbooks and dictionaries consists of three conjugation groups. The significant difference from the traditional model is that this model is based on

Roman alphabet, which indicates individual phonemes rather than syllables. According to the way they are conjugated by spelling out with Roman letters, the morphological verb conjugation model defines two regular conjugation groups; one is called the "Consonant Ending Verb" group and is derived from the "Five Grade Conjugation" pattern in the traditional model, and the other is the "Vowel Ending Verb" group including the "Upper One-tier" and "Lower One-tier" Conjugation patterns in the traditional model. Thus, in this model, there are two regular verb conjugation groups and one irregular verb conjugation group that contains only two verbs of *"kuru"* and *"suru"* as mentioned in the traditional model.

In the "Consonant Ending Verb" pattern, the base verb in "Five Grade Conjugation" is split into two portions, a primary base verb and the followed vowel; for example, the imperative form of *"kaku"* consists of a verb base *"kak"* and the followed vowel *'u.'* The reason for the disjunction is because *"kak"* is a fixed form in all forms but only those following vowels are altered in the conjugation. The disjunction of a base verb results in a primary base verb ending with a consonant; therefore, it is called a "Consonant Ending Verb". However, the ending vowels in both "Upper One-tier" and "Lower One-tier" conjugation patterns do not change according to its different forms. The verb conjugated ending in both "Upper One-tier conjugation" and "Lower One-tier conjugation" are fixed vowels such as ***'i'*** and ***'e'*** in the verbs ***"oki**ru"* and ***"tabe**ru"*** as in Table 2 and 3. Thus, both patterns are called "Vowel Ending Verbs."

In sum, I will utilize this morphological verb conjugation model because it is commonly known among non-native Japanese learners and the Prolog program uses Roman alphabetic letters not Japanese syllabary letters. And I define these three grouping names in the morphological model hereafter as consonant verbs, vowel verbs, and kuru-irregular verb and suru-irregular verb respectively in this paper and the Prolog programming.

2. LANGUAGE STRUCTURE ANALYSIS

In this section, I will analyze the structure of a Japanese verb phrase based on the word formation rules and the process of verb formation to facilitate a better understanding of the Japanese verb conjugation process for Prolog programming.

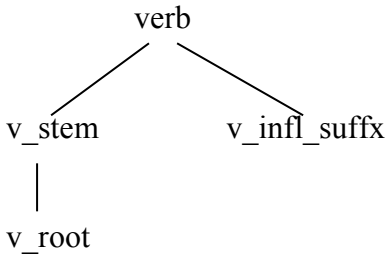
2.1. Word Formation Rule Analysis

The word formation rules establish the mother-daughter-sister relations among the relevant lexical items. Following Sobin (2013), word formation rules such as the ones below can be written to account for the hierarchic structure of complex words. He defines English verb formation rules in his Prolog program "all_paradigms.pl" as follows:

verb - - > vstem, vsuf.
vstem - - > vroot (Sobin, 2013).

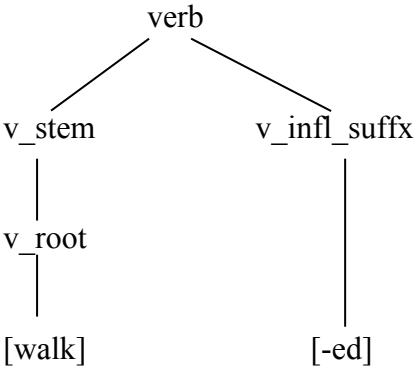
First of all, these rules involve terminology that is important to understand for a morphological analysis as well as for a Prolog program. The arrow '- - >' means "consists of" or "contains." So, the first rule says that "a verb contains a sequence of a verb stem and verb suffixes." Also, some prominent components, a root and a suffix, are required to build the verb form. In morphology, these two elements are called morphemes. A root is the core morpheme that a word is built on. A suffix is a non-root, which is attached to a stem to construct a word. The difference between suffixes and other affixes such as prefixes and infixes is that suffixes are always attached after a root. The suffixes are of two types: derivational suffixes and inflectional suffixes. While derivational suffixes are attached to a stem to create new or different words; inflectional suffixes are not used to create new words, but to add information to words or to create different forms of the word in the same word class. In English verb formation, an inflectional suffix attaches to a verb stem to add information to the verb such as tense. For example, the verb "walked" is broken down into two portions as "walk" as a verb root and "-ed" as an inflectional suffix referring to tense. The following figure Tree 1 shows the English verb formation in a tree structure.

[Tree 1] Tree Structure of English Verb Formation



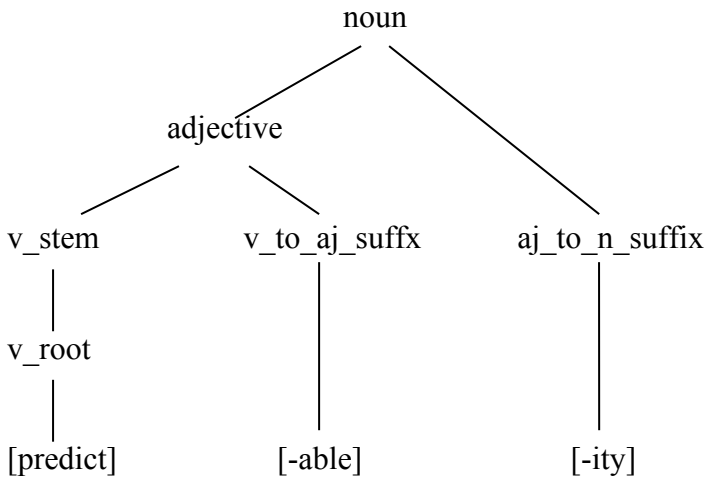
The tree structure of the example verb “walked” is shown in Tree 2.

[Tree 2] Tree Structure of the Verb “walked”



A derivational suffix, which attaches to a stem, functions to create a new or different grammatical category. For example, the verb “predict” can be derivated to two different grammatical categories, adjective and noun, by attaching derivational suffixes. It becomes the adjective “predictable” by attaching the adjective derivational suffix “-able” to the verb stem. The noun form of the verb “predict” has two derivational ways. One is to create the noun “prediction” by attaching the noun derivational suffix “-ion” to the verb stem and the other is to create the noun “predictability” derived from the adjective “predictable” by adding noun derivational suffix “-ity” to the adjective. The two-step derivational word formation of “predictability” is represented in Tree 3 below.

[Tree 3] Derivational Tree Structure of the noun "predictability" from the verb "predict"



The derivational tree structure analysis proves that English word formation is constructed in a binary hierarchy structure. If the word formation is constructed in a single linear structure, it would not enable words to derive from one grammatical category to the others to create such complex words. According to the word formation rule analysis, Japanese verb formation is also described with a root, a stem, and auxiliary suffixes in a hierarchy structure.

2.2 Japanese Verb Formation Analysis

Japanese verb formation can be described in a word formation rule analysis like English verbs. According to Iwasaki (2013:78), "[t]he root is followed by a stem forming suffix (SFS) to form a stem… [a] stem is followed by one or more auxiliary suffixes (Aux Suffixes) to construct a verb form, as shown below" and he provides a verb formation below:

←----------VERB FORM----------→
←----------STEM---------→
[[ROOT = SFS] -Aux Suffix 1 … -Aux Suffix n]

(Iwasaki, 2013:79).

In applying this model, the polite past form of the verb ***"kaku," "kaki-mashi-ta"*** would be shown as follows:

[verb [v-stem[v-root ***kak*]** ***-i***(sfs)] ***-mashi*** Aux Suffix1(polite form), ***-ta*** Aux Suffix2 (past form)]

According to the Iwasaki's (2013) Japanese verb formation, a Japanese verb form seems to be constructed by attaching sequence of auxiliary suffixes to a verb stem in a linear structure as shown below.

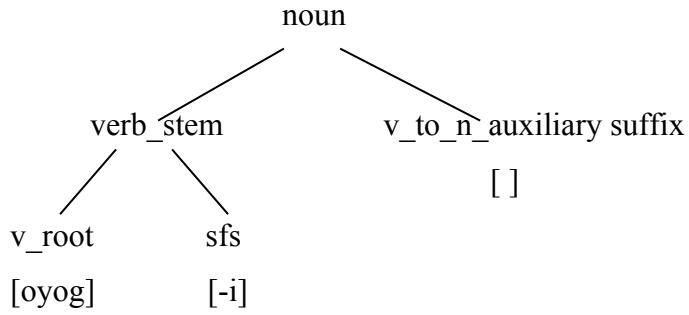
verb → verb_stem, auxiliary suffix1, auxiliary suffix 2, …, auxiliary suffix n.

verb_stem → verb_root, SFS.

However, I analyze Japanese word formation as a binary hierarchy structure like English word formation. The derivational word formation of a Japanese word also shows the evidence of its binary hierarchy structure. There are three different ways to derive a verb to a noun in Japanese: Zero suffix nominalization, Causative/Passive-like suffix nominalization, and Noun suffix nominalization.

Zero suffix nominalization takes place by attaching a null auxiliary suffix to a verb stem in order to derive a noun from a verb. Volpe (2005:35) introduces this type of nominalization in Japanese as follows. "One type is morphologically zero-related to the verb, or more specifically, to the verbal stem, called *renyōkei* in Japanese. An example is the nominalization *oyogi* 'swimming,' etymologically-related to the verb *oyog-u* 'swim-NON-PAST'." The below Tree 4 shows how the noun "*oyogi*" is derived from the verb "*oyogu*."

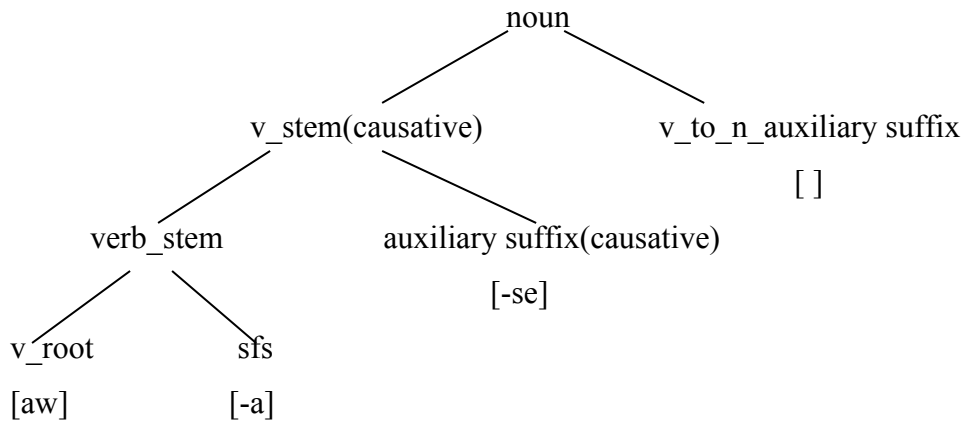
[Tree 4] Derivational Tree Structure of the noun *"oyogi"* from the verb *"oyogu"*



Causative/Passive-like suffix nominalization occurs at a higher level than Zero suffix nominalization in the hierarchy structure of a verb formation. Volpe (2005:43) introduces the Causative/Passive-like suffix nominalization as an arbitrary semantic related nominalization. "The [verb] root *aw-* of the paired verbs *a-u* 'meet'/*aw-ase-ru* 'join' together with the causative morpheme -[*a*]*se,* yields the etymologically-related nominalization *awase* 'a lined kimono.' […] *Awase* is not 'a joined thing,' but is associated only with 'a kimono (that results from joining it with a lining)'." Tree 5 shows the derivation of the

Causative/Passive- like suffix nominalization of the noun "*awase*."

[Tree 5] Derivational Tree Structure of the noun *"awase"* from the verb *"au"*

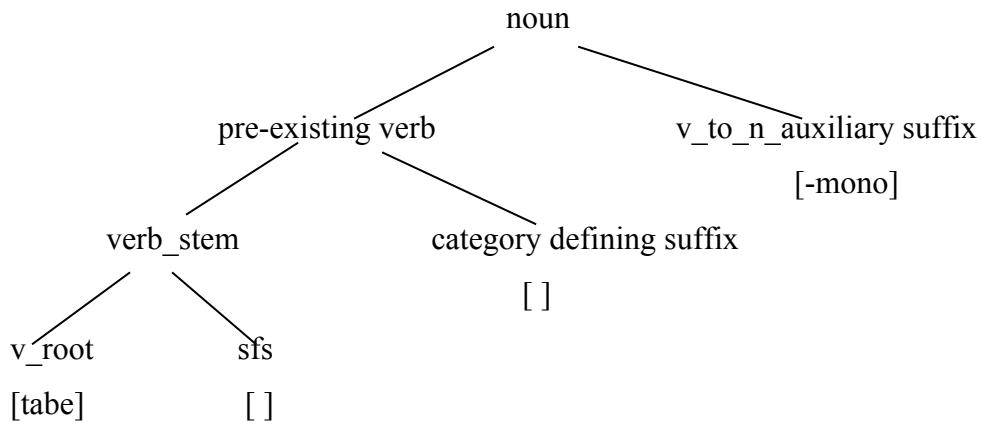


An auxiliary suffix for nominalization in Japanese is not always null as you see the above. In Noun suffix nominalization, there are several noun suffixes to be used such as "*-mono*" implying "thing/person" and "*-kata*" implying "the way." Volpe (2005:63) addresses the Noun suffix nominalization as Deveral *mono* nominalization.

> The meaning of *tabe-mono*, from stem of *tabe-ru* 'eat' can be paraphrased as 'something that is eaten,' i.e., 'food.' Such nominalizations are a case of word-formation from pre-existing words; that is, a noun is formed from a pre-existing verb.

Tree 6 shows the derivation of the Noun suffix nominalization of the noun "*tabe-mono.*"

[Tree 6] Derivational Tree Structure of the noun *"tabe-mono"*



Thus, these three derivational nominalizations in Japanese indicate ~~t~~hat Japanese verb

formation is also constructed in a binary hierarchy structure as seen in English word formation. Based on the word formation analysis, the word formation rule of Japanese verbs should be defined in a hierarchy structure as well.

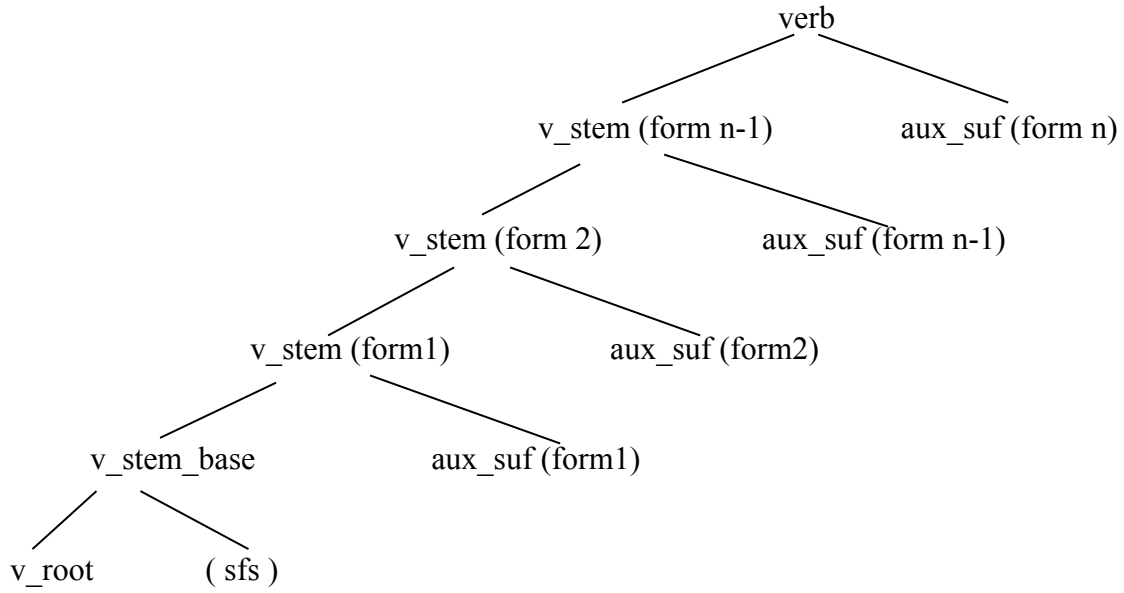
2.3 Word Formation Rule of Japanese Verbs

According to the word formation analysis of Japanese verb conjugation discussed in Section 2.2, the sequence of auxiliary suffixes are not attached as in a linear structure but the various auxiliary suffixes attach to a verb stem recursively to form a verb. By this definition, the word formation rule in Japanese verb formation is defined as follows:

verb → v_stem(form n-1), aux_suf(form n).
v_stem(form n-1) → v_stem(form 2), aux_suf(form n-1).
v_stem(form 2) → v_stem(form 1), aux_suf(form 2).
v_stem(form 1) → v_stem_base, aux_suf(form 1).
v_stem_base → v_root, sfs.

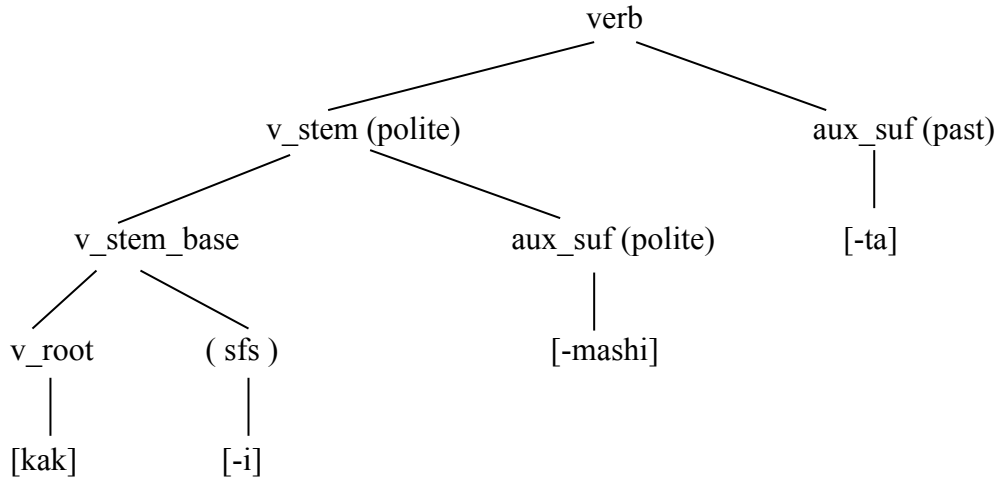
The tree structure by following the above rules is shown in Tree 7.

[Tree 7] Tree Structure of Japanese Verb Formation



Then, Tree 8 represents the polite past form of the verb *'kaku,'* ***"kak-i-mashi-ta"*** with the word formation rule.

[Tree 8] Tree Structure of Polite Past Form of the Verb *"kaku"*



As discussed the above, Japanese verb formation is constructed similarly to English word formation. The difference is that an English inflectional suffix is replaced with an auxiliary suffix, which follows the verb stem to derive various forms of a verb. That means that the auxiliary suffixes mostly behave similarly to English inflectional suffixes but they carry various semantic functions and are able to stack up with other suffixes. Consequently, this large functional variation of auxiliary suffixes in Japanese verb conjugation parallels what English does with auxiliary verbs.

3. JAPANESE VERB CONJUGATION ANALYSIS

In this section, I discuss the theoretical details of the three Japanese verb conjugation groups: consonant verbs, vowel verbs, and irregular verbs from the analysis of 306 Japanese verbs in *600 Basic Japanese Verbs* (2013) and *501 Japanese verbs* (1998) with respect to verb forms, the patterns of each conjugation group, and the irregularity in the conjugation. This pattern analysis of Japanese verb conjugation will guide the setup of the prolog programming.

3.1 Verb Forms

Before the discussion of each verb conjugation group, it is necessary to discuss the forms included in most of the verb conjugation dictionaries. It is important to understand that verb stems are augmented by the forms of auxiliary suffixes. I introduce eleven fundamental forms in this section: imperfective-form(未然形), continuative-form(連用形), past-tense-form(過去形), polite-form(丁寧形), Predicative-form(終止形), causative-form(使役形), passive-form(受身形), potential-form(可能形), imperative-form(命令形), conditional-form (仮定形); and volitional-form(意向形).

3.1.1 Imperfective Form/未然形(mizen kei)

The imperfective form, which is called *mizen kei* 未然形 in Japanese, indicates negation in a verb formation. This form provides a verb with various negative forms attached to the negative auxiliary suffixes beginning with ***'-na.'*** Below is the list of the negative forms with applicable auxiliary suffixes.

Informal present negative form: ***-nai***

(e.g. '*kak-a-**nai***' meaning '[S] do(es) not write')

Informal past negative form: ***-nakatta***

(e.g. '*kak-a-**nakatta***' meaning '[S] did not write')

Informal present negative conditional form: ***-nakereba***

(e.g. '*kak-a-**nakereba***' meaning 'if [S] do(es) not write')

Informal past negative conditional form: ***-nakattara***

(e.g. '*kak-a-**nakattara***' meaning 'if [S] did not write')

Informal present negative continuative form: ***-nakute***

(e.g. '*kak-a-**nakute***' meaning '[S] is not writing')

3.1.2 Continuative Form/連用形(renyō kei)

The continuative form is called *renyō kei* 連用形 in Japanese and sometimes introduced as "*Te* form" in some grammar textbooks. Generally, this form is considered to be equivalent to the English progressive It is followed by the continuative auxiliary suffixes ***-te/-de*** to express present progress, present perfect, and event sequences. This form often causes morphophonemic alternations in verb conjugation. I will discuss this Japanese morphophonemic irregularity in detail in Section 4 later.

3.1.3 Past Tense Form/過去形(kako kei)

The past tense form is called *kako kei* 過去形 in Japanese and also is introduced as "*Ta* form" in some grammar textbooks. It behaves similarly to the continuative form with the past tense auxiliary suffixes ***-ta/da*** in terms of morphophonemic alternations

3.1.4 Polite Form/丁寧形(teinei kei)

The polite-form is called *teinei kei* 丁寧形 and is often referred to as "*masu* form" in some grammar books since it is followed by the formal auxiliary suffix ***-masu***. This form generally compares to the English present and future forms. It is a crucial form in Japanese culture that puts high emphasis on the value of politeness or formality. Generally, the polite form is used in written context and when spoken, is distinguished by specific situations and relationships in which the speaker must show respect.

In terms of verb formation, the polite form generates several different formal expressions with the various auxiliary suffixes beginning with *'-mas'* as shown the below.

Formal present affirmative form: ***-masu***
Formal past affirmative form: ***-mashita***
Formal present negative form: ***-masen***
Formal past negative form: ***-masendeshita***
Formal past affirmative conditional form: ***- mashitara***
Formal past negative conditional form: ***-masendeshitara***
Formal present affirmative volitional form: ***-mashō***

3.1.5 Predicative Form/終止形(shūshi kei)

The 'base' or 'simple' form of the verb is called the 'predicative' form in some grammar books. This form is important for non-native Japanese learners because it is the

form found in dictionaries. In Japanese traditional grammar, it is called *shūshi kei* 終止形, which literally means "end-form" because this form is the complete verb form without any auxiliary suffixes.

3.1.6 Causative Form/使役形(shieki kei) and Passive Form/受身形(ukemi kei)

These two forms share semantic function with English causative and passive forms, respectively. Each of them are introduced as 使役形 (shieki kei) for the causative form and 受身形 (ukemi kei) for the passive form in Japanese. The causative verb form is constructed with the auxiliary suffixes either ***-saseru*** or ***-seru*** depending on the preceding verb conjugation groups: ***-saseru*** is attached to the stem of a vowel verb and kuru irregular verb, and ***-seru*** is attached to the stem of a consonant verb and suru irregular verb. The auxiliary suffixes for passive form also represent in two suffixes: ***-rareru*** or ***-reru***. The suffix ***-rareru*** attaches to the stem of a vowel verb and kuru irregular verb, and ***-reru*** attaches to the stem of a consonant verb and suru irregular verb.

3.1.7 Potential Form/可能形(kanou kei)

This form expresses capability or possibility with four different auxiliary suffixes. It is introduced as 可能形 (kanou kei) in Japanese. Unlike the other forms, the potential form carries four auxiliary suffix variations: ***-rareru***, ***-reru,*** ***-ru***, and ***-kiru***. A vowel verb generates two Potential forms attached with ***-rareru*** and ***-reru***. A consonant verb generates one form with ***-ru***. And two irregular verbs are followed by the different suffixes correspondingly: ***-reru*** for kuru-irregular verb and ***-kiru*** for suru-irregular verb.

3.1.8 Imperative Form/命令形(meirei kei)

The imperative form is introduced as 命令形 (meirei kei) in Japanese. There are two imperative forms with the corresponding various auxiliary suffixes: the informal affirmative imperative form and the informal negative imperative form. The informal affirmative imperative form carries three suffixes: **-[]** for a consonant verb, ***-ro*** for a vowel verb and suru-irregular verb, and ***-i*** for kuru-irregular verb. The informal negative imperative form carries two suffixes: ***-na*** for a consonant verb and ***-runa*** for a vowel verb and suru- and kuru-irregular verbs.

3.1.9 Conditional Form/仮定形(katei kei) and Volitional Form/意向形(ikou kei)

The conditional form is the equivalent to the English "if-form" called 仮定形 (katei kei) in Japanese. The volitional form compares to the English "let's-form" called 意向形 (ikou kei) in Japanese. Both forms similarly possess two auxiliary suffixes depending on each conjugation group. In the conditional form, the suffix ***-reba*** is used for a vowel verb and the two suru- and kuru- irregular verbs, and the suffix ***-ba*** is used for a consonant verb. The suffixes attach on each group's verb stems to derive the informal present affirmative conditional form. The informal past affirmative conditional form is expressed with the auxiliary suffix ***-ra*** such as 'kai-***ta-ra***' meaning "if [S] wrote" or 'tabe-***ta-ra***' meaning "if [S] ate." In the volitional form, the suffix ***-yō*** for a vowel verb and two irregular verbs and the suffix ***-o*** for a consonant verb attach on the corresponding verb stem to derive the informal present affirmative volitional form.

3.2 Verb Conjugation Groups

Japanese verbs are classified into three conjugation groups: consonant verbs, vowel verbs, and irregular verbs. It is very easy to learn irregular verbs in Japanese because there are only two verbs, *"kuru"* (to come) as the "kuru-irregular verb" and *"suru"* (to do) as the "suru-irregular verb." It is also possible to predict each conjugation group, either a consonant verb or a vowel verb, according to the verb root ending rules. Iwasaki (2013:78) found the specific root ending to define each verb conjugation group such that "[t]he root of consonant verbs ends in one of the following consonants, *k, s, t, r, b, g, m, n* or the semi-vowel *w,* … In contrast, the vowel verb root ends in either *e* or *i.*"

As mentioned in Section 2, the Japanese verb form is constructed with a verb stem, which consists of a verb root and a stem forming suffix (SFS), and auxiliary suffixes. The SFS is only required for consonant verbs to fulfill the typical Japanese phonological sequence pattern as *CVCV* in the process of their conjugation. Sasaki (2008:9) discusses the requirement of SFS for consonant verbs in their verb conjugation as follows:

> The [former] two vowels, /i/ and /e/, are the stem-final vowels of vowel verbs and the [last] vowel /a/ is the epenthetic vowel used to avoid an unwelcome double consonant sequence resulting from the stem-final consonant of a consonant verb followed by the initial consonant of the [negative] suffix.

In order to follow the Japanese *CVCV* phonotactic rule, consonant verbs must be followed by an epenthetic vowel. The SFS breaks up consonant cluster.

[Table 6] Verb Paradigm of the Consonant Verb *"kaku"*

	[Root]	[SFS]	[Aux Suffixes]
<Imperfective form>	kak	**-a**	-nai
<Causative form>	kak	**-a**	-seru
<Passive form>	kak	**-a**	-reru
<Polite form>	kak	**-i**	-masu
<Continuative form>	kak	**-i**	-te
<Past-tense form>	kak	**-i**	-ta
<Predicative form>	kak	**-u**	-[]
<Negative imperative form>	kak	**-u**	-na
<Conditional form>	kak	**-e**	-ba
<Imperative form>	kak	**-e**	-[]
<Potential form>	kak	**-e**	-ru
<Volitional form>	kak	**-o**	-o

While consonant verbs require SFSs in their verb formation, vowel verbs and irregular verbs require an extra syllable as allomorphy in some auxiliary suffixes. McCawley (1968:95) also notices the emergence of allomorphy in a vowel verb conjugation that is unlike what happens with a consonant verb. He writes "the vowel-stem verbs have affixes beginning with /r,s,y/: *-ru* for predicative form, *-reba* for conditional form, *-rare* for passive form, *-sase* for causative form, and *-yoo* for volitional form." These allomorphy rules are shown in bold italic letters in the verb paradigms of vowel verbs and two irregular verbs in Table 7, 8, 9.

[Table 7] Verb Paradigm of the Vowel Verb *"taberu"*

	[Root]	[SFS]	[Aux Suffixes]
<Imperfective form>	tabe	-[]	-nai
<Causative form>	tabe	-[]	-***sa***seru
<Passive form>	tabe	-[]	-***ra***reru
<Polite form>	tabe	-[]	-masu
<Continuative form>	tabe	-[]	-te
<Past-tense form>	tabe	-[]	-ta
<Predicative form>	tabe	-[]	-***ru***
<Negative imperative form>	tabe	-[]	-***ru***na
<Conditional form>	tabe	-[]	-***re***ba
<Imperative form>	tabe	-[]	-***ro***
<Potential form>	tabe	-[]	-***re***ru
<Volitional form>	tabe	-[]	-***yo***o

[Table 8] Verb Paradigm of the *"kuru"* Irregular Verb

	[Root]	[SFS]	[Aux Suffixes]
<Imperfective form>	**ko**	-[]	-nai
<Causative form>	**ko**	-[]	-***sa***seru
<Passive form>	**ko**	-[]	-***ra***reru
<Polite form>	**ki**	-[]	-masu
<Continuative form>	**ki**	-[]	-te
<Past-tense form>	**ki**	-[]	-ta
<Predicative form>	**ku**	-[]	-***ru***
<Negative imperative form>	**ku**	-[]	-***ru***na
<Conditional form>	**ko**	-[]	-***re***ba
<Imperative form>	**ko**	-[]	-**i**
<Potential form>	**ko**	-[]	-***re***ru
<Volitional form>	**ko**	-[]	-***yo***o

[Table 9] Verb Paradigm of the *"suru"* Irregular Verb

	[Root]	[SFS]	[Aux Suffixes]
<Imperfective form>	**shi**	-[]	-nai
<Causative form>	**sa**	-[]	-seru
<Passive form>	**sa**	-[]	-reru
<Polite form>	**shi**	-[]	-masu
<Continuative form>	**shi**	-[]	-te
<Past-tense form>	**shi**	-[]	-ta
<Predicative form>	**su**	-[]	***-ru***
<Negative imperative form>	**su**	-[]	***-ru***na
<Conditional form>	**su**	-[]	***-re***ba
<Imperative form>	**shi**	-[]	***-ro***
<Potential form>	**de**	-[]	***-ki***ru
<Volitional form>	**shi**	-[]	***-yo***o

Moreover, notice that each of two irregular verbs carries more than two verb root variations: *kuru*-irregular verb has three roots as *ko, ki,* and *ku*, and *suru*-irregular verb has four roots as *sa, shi, su,* and *de,* respectively.

4. MORPHOPHONEMIC PROCESS: ONBIN

In the analysis of 304 regular verbs, I found some morphophonemic alternations in their verb stem conjugations. The typical Japanese morphophonemic alternation observed in the verb conjugation is called *Onbin* (音便) in Japanese. Iwasaki (2013:83) defines the *Onbin* in his book, *Japanese*, as follows: "This [*Onbin*] is usually translated as "sandhi" or "sound euphony," a morphophonemic process in which the sound is modified under the influence of an adjacent sound for easier and/or pleasing pronunciation." The *Onbin* Processes are found only on the conjugation of consonant verbs, specifically in the continuative and past-tense forms with *-t/-d* initial auxiliary suffixes attached. I observed three main *Onbin* Processes that were found in the specific ending phonemes of consonant verbs from the analysis.

Process 1 occurs with the consonant verbs ending root in *-t, -r, -u*.
Process 2 occurs with the ones ending root in *-n, -m, -b*.
Process 3 occurs with the ones ending root in *-k, -g*.

In addition to these three *Onbin* morphophonemic phenomena, I also found minor alternations. In Process 1, I found the consonant */w/* insertion between vowel clusters in the imperfective forms, such as *"arasowanai"* meaning "not to fight." Also, the ending phoneme *-t* of a verb root such as "*kat"* in the verb "*katsu"* (to win) changes under the influence of the initial phonemes in the followed auxiliary suffixes. The phoneme *-t* becomes *-ts* when it is followed by the stem forming suffix *-u* and the phoneme *-t* becomes *-ch* when it is followed by the stem forming suffix *-i*. In Process 3, I also noticed the phoneme *-s* becomes *-sh* when it is followed by the stem forming suffix *-i*. Pearson (1972:95) discusses these Japanese dental alternations: /t/ → /ts/, /t/ → /tš/, and /s/ → /š/. He states that the phonemic alternations of [t,s] → [tš,š]/__ i are proceeded by palatalization and affrication; and the other phonemic alternation of [t] → [ts]/__ u is proceeded by affrication. These minor phonemic alternations are also included in the Prolog programming.

4.1. *Onbin* Process 1

The first *Onbin* Process occurs when the root of consonant verbs ends in *-t, -r, -u(w)*. Iwasaki (2013:83) addresses the regularity of the *Onbin* Process as follows: "(W)ith roots ending in *-t, -r, -(w)*, the stem forming suffix, *-i-*, elides, then the final consonant of the root assimilates to the *t*, yielding a geminate consonant," and he also defines the morphophonemic process of *Onbin* Process 1 with example verbs: "*katsu*" meaning "to win" for *-t* root ending, "*ageru*" meaning "to go up" for *-r* root ending, and "*au*" meaning "to meet" for *-u(w)* root

ending in Table 10.

[Table 10] ***"Onbin"*** **Process 1 for Consonant Verbs Ending Root in** ***-t, -r, -u(w)***

[Rule-based formation]	*ka**ch**-**i**-te (katsu)*	*agar-**i**-te (agaru)*	*a-**i**-te (au)*
[*"Onbin"* process]			
i **deletion**	*ka**ch**_ -te*	*aga**r**_ -te*	*a**(w)**_ -te*
Assimilation	*ka**t** -te*	*aga**t** -te*	*a**t** -te*

(Iwasaki, 2013:84)

In order to recognize these *Onbin* Processes in a Prolog program, I define two root variations in the consonant root ending *-r* and *-u(w)* verbs; *-t* root ending consonant verbs do not carry the root variation for this *Onbin* Process because this verb root originally ends in *-t*. Each of the applied consonant verbs uses a regular conjugation root and the alternative root for the *Onbin* Process to compute each form of the applicable verbs precisely. The sample consonant verb roots are explained as follows:

- Consonant verbs ending root in *-r*:
  e.g. the roots of *agaru* are *ager* and *agat (aga**r** → aga**t**)*
- Consonant verbs ending root in *-u(w)*:
  e.g. the roots of *au* are *a(w)* and *at (a**(w)** → a**t**)*

Out of 197 consonant verbs, I found 106 verbs (7 *-t* root ending verbs, 71 *-r* root ending verbs, and 28 *-u(w)* root ending verbs) belonging to the *Onbin* Process 1.

Moreover, as mentioned above, only *-tsu* ending consonant verbs, such as *"katsu,"* have additional morphophonemic alternations involved in its conjugation. As in Table 10, the rule-based formation of *"katsu"* in the continuative form is *kachi-te* not *kati-te*. The root ending *-t* has been changed into *-ch* by being influenced with the followed phonemic /i/. These *Onbin* Processes in the verb *"katsu"* conjugation are observed in Table 11.

[Table 11] Verb Paradigm for the Consonant Verb *"katsu"*

	[Root]	[SFS]	[Aux Suffixes]
<Imperfective form>	kat	**-a**	-nai
<Causative form>	kat	**-a**	-seru
<Passive form>	kat	**-a**	-reru
<Polite form>	**ka*ch***	**-i**	-masu
<Continuative form>	ka~~***cht***~~	~~**-i**~~	-te
<Past-tense form>	ka~~***cht***~~	~~**-i**~~	-ta
<Predicative form>	**ka*ts***	**-u**	-[]
<Negative imperative form>	**ka*ts***	**-u**	-na
<Conditional form>	kat	**-e**	-ba
<Imperative form>	kat	**-e**	-[]
<Potential form>	kat	**-e**	-ru
<Volitional form>	kat	**-o**	-o

In the above Table 11, the verb *"katsu"* carries three root variations: *'kat,' 'kach'* and *'kats.'* The root variations, *'kach'* and *'kats,'* result from the minor morphophonemic alternations introduced before the phoneme /t/ is palatalized and affricated to become /tš/ as *-ch* because it is influenced by the followed vowel /i/. When followed by the vowel /u/, the phoneme /t/ is affricated to become /ts/ as *-ts*. Thus, for those *tsu* ending consonant verbs I define three distinct roots, like *'kat,' 'kats,'* and *'kach'* for *"katsu,"* to compute their conjugation properly in the program.

Furthermore, the vowel cluster verb *"au"* meaning "to meet" (see Table 10) also shows the other morphophonemic phenomenon. Because the verb is constructed in a sequence of two vowels, the verb root ends in the vowel '*a*' even though it is classified as a consonant verb. The vowel cluster verb *"au"* also creates another vowel cluster with SFS in the conjugation. Some of the vowel combinations such as the vowel cluster '*aa*,' do not work phonetically in Japanese. The vowel cluster '*aa*' in the verb *"au"* happens in imperative, causative, and passive forms with their SFS ***-a***. To avoid these inappropriate vowel clusters, the phoneme /w/ is inserted after the root ending vowel '*a*' so that its imperfective form becomes *'awanai'* instead of *'aanai.'* This morphophonemic phenomenon is known as a consonant insertion. The consonant /w/, which sounds close to the vowel /u/ is inserted to break up the vowel cluster. This insertion occurs in the consonant verbs ending in *VV*

sequence such as *"kau"* meaning "to buy," *"arasou"* meaning "to fight," and *"iu"* meaning "to say."

4.2. *Onbin* Process 2

Onbin Process 2 is observed when the root of consonant verbs ends in *-n, -m,* and *-b*. These verb root ending phonemes /n/, /m/, and /b/ are nasalized to become /n/ when they attach to /t/ beginning auxiliary suffixes *-te* in the continuative form and *-ta* in the past tense form. In this process, these initial auxiliary suffixes *-te* and *-ta* are assimilated to the proceeding /n/ to become *-de* and *-da*, respectively (Iwasaki, 2013:84). Iwasaki also discusses the *Onbin* Process 2 with the sample verbs *"shinu"* meaning "to die" for *-n* root ending verb, *"nomu"* meaning "to drink" for *-m* root ending verb, and *"asobu"* meaning "to play" for *-b* root ending verb in Table 12.

[Table 12] *"Onbin"* Process 2 for Consonant Verbs Ending Root in *-n, -m, -b*

[Rule-based formation]	*shin-**i**-te (shinu)*	*nom-**i**-te (nomu)*	*asob-**i**-te (asobu)*
[*"Onbin"* process]			
***i* deletion**	*shi**n**_ -te*	*no**m**_ -te*	*aso**b**_ -te*
Nasalization	-	-	*aso**m** -te*
Assimilation	-	*no**n** -te*	*aso**n** -te*
Voicing	*shi**n** -de*	*no**n** -de*	*aso**n** -de*

(Iwasaki, 2013:84)

As in *Onbin* Process 1, the consonant verbs with root ending in *-m,* and *-b* require two root varieties to compute their conjugation in the program. For example, the root ending in *-m* of *"nomu"* carries the two root varieties *'nom'* and *'non,'* and the root ending in *-b* of *"asobu"* carries *'asob'* and *'ason.'* The root ending in *-n* of *"shinu"* carries just one root: *'shin.'* Out of 197 consonant verbs, I counted 24 *Onbin* Process 2 consonant verbs (1 *-n* root ending verb, 14 *-m* root ending verbs, and 9 *-b* root ending verbs).

4.3. *Onbin* Process 3

The third *Onbin* Process occurs with the consonant verbs ending in the roots *-k* and *-g*. When the phoneme /i/ attaches to the *k* and *g* ending consonants, palatalization occurs to change the roots to k^y and g^y. Afterwards, both of the palatalized consonants are assimilated to become /*i*/ by vowelization. Iwasaki (2013:84) describes the *Onbin* Process 3 and shows the morphophonemic process with the example verbs, *"aku"* meaning "to open" as for *-k*

root ending verb and *"oyogu"* meaning "to swim" as for *-g* root ending verb, in Table 13:

[Table 13] *"Onbin"* Process 3 for Consonant Verbs Ending Root in *-k, -g*

[Rule-based formation]	*ak-**i**-te (aku)*	*oyog-**i**-te (oyogu)*
[*"Onbin"* process]		
Palatalization	*a**k**^{y} -**i**-te*	*oyo**g**^{y} -**i**-te*
***i* deletion**	*a**k**^{y} _ -te*	*oyo**g**^{y} _ -te*
Voicing	**-**	*oyo**g**^{y} _ -**d**e*
Vowelization	*a**i** -te*	*oyo**i** –de*

(Iwasaki, 2013:84)

This *Onbin* Process also requires two root varieties for each verb. In the above examples, the verb *"aku"* carries *'ak'* and *'ai,'* and the verb *"oyogu"* carries *'oyog'* and *'oyoi.'* There are 63 *-k* root ending verbs and 4 *-g* root ending verbs for a total of 67 *Onbin* Process 3 verbs.

Moreover, as mentioned at the beginning of this section, *Onbin* Process 3 includes a minor alternation of /s/ → /š/. These phonemes represent the same letter し, in Japanese syllabary writing system. Native Japanese speakers would not be able to distinguish the abstract sound difference between /s/ as in *si* and /š/ as in *shi* in their speech because these sounds exist only in the mind of native Japanese speakers. These sounds would be caused by the speaker's "psychological reality" (Bermúdez-Otero, 2012). While both of them are written in one syllabary letter し in Japanese, in Roman alphabetical writing system they are distinguished with *si* and *shi*. Cole and Hualde (2011:8) discuss the sound abstractness of Japanese *si* and *shi*:

> For the Japanese case, an Abstract Phonemic analysis posits the phoneme /s/, relegating [ʃ] to the status of an allophone: /s/ maps onto the allophone [ʃ] in surface realization when it precedes phonemic /i/ and also before the glide /j/, a kind of 'ghost' phoneme that serves to condition the palatal sibilant and is simultaneously absorbed into that consonant.

According to Cole and Hualde, two abstract phonemes exist in the sound of し in Japanese, though they are hardly distinguishable sounds for native Japanese speakers because these sounds exist in a speakers' psychological reality. Thus, I standardize these two abstract phonemes as "*shi*" in my paper and programming.

These *Onbin* Processes in Japanese verb conjugation show interesting morphophonemic traits in Japanese as a syllabic language. I include these irregularities of the verb conjugation in the program to derive each verb forms precisely.

5. AUXILIARY SUFFIXES IN VERB FORMATION

Besides a verb stem conjugation, there are various semantic functional auxiliary suffixes to construct the verb forms in Japanese. According to Iwasaki (2013:81-83), they are categorized into four different groups: *voice derivational suffix(VDS), termination suffix 1(TS1), termination suffix 2(TS2),* and *conjunctive suffix(CS)*. He states that *VDS* follows right after the conjugated verbs referring to causative, passive, desiderative, and difficulty; *TS1* and *TS2* come at the end of sentences. One of the major differences in these two suffixes is that *TS1* can be further suffixed, but *TS2* cannot. Functionally, *TS1* carries tense and polarity, while *TS2* carries aspect, mood and modality. *CS* follows a stem to form various adverbial clauses. In sum, these different suffixes can concatenate to form a long verb form.

However, from the analysis of auxiliary suffix formation, I categorized them into five different groups accordingly to each semantic function: *Derivational Suffix (DS), Formality, Polarity, Tense,* and *Conjunction.* They are lined up in the certain sequence to construct the phrases in the verb formation.

[Table 14] The Sequence of Auxiliary Suffixes

[Derivational Suffix]	**→ [Formality]**	**→ [Polarity]**
-*re/rare* (Passive)	-*mas* (Formal present)	-[] (Affirmative)
-*se/sase* (Causative)	-*mashi* (Formal past)	-*na/n* (Negative present)
-*re/rare* (Potential)	-*mase* (Formal negative)	-*nakat* (Negative past/conditional)
-[] (Other forms)	-*masho* (Formal volitional)	-*naku* (Negative continuative)
	-[](Informal)	-*nake* (Negative conditional)

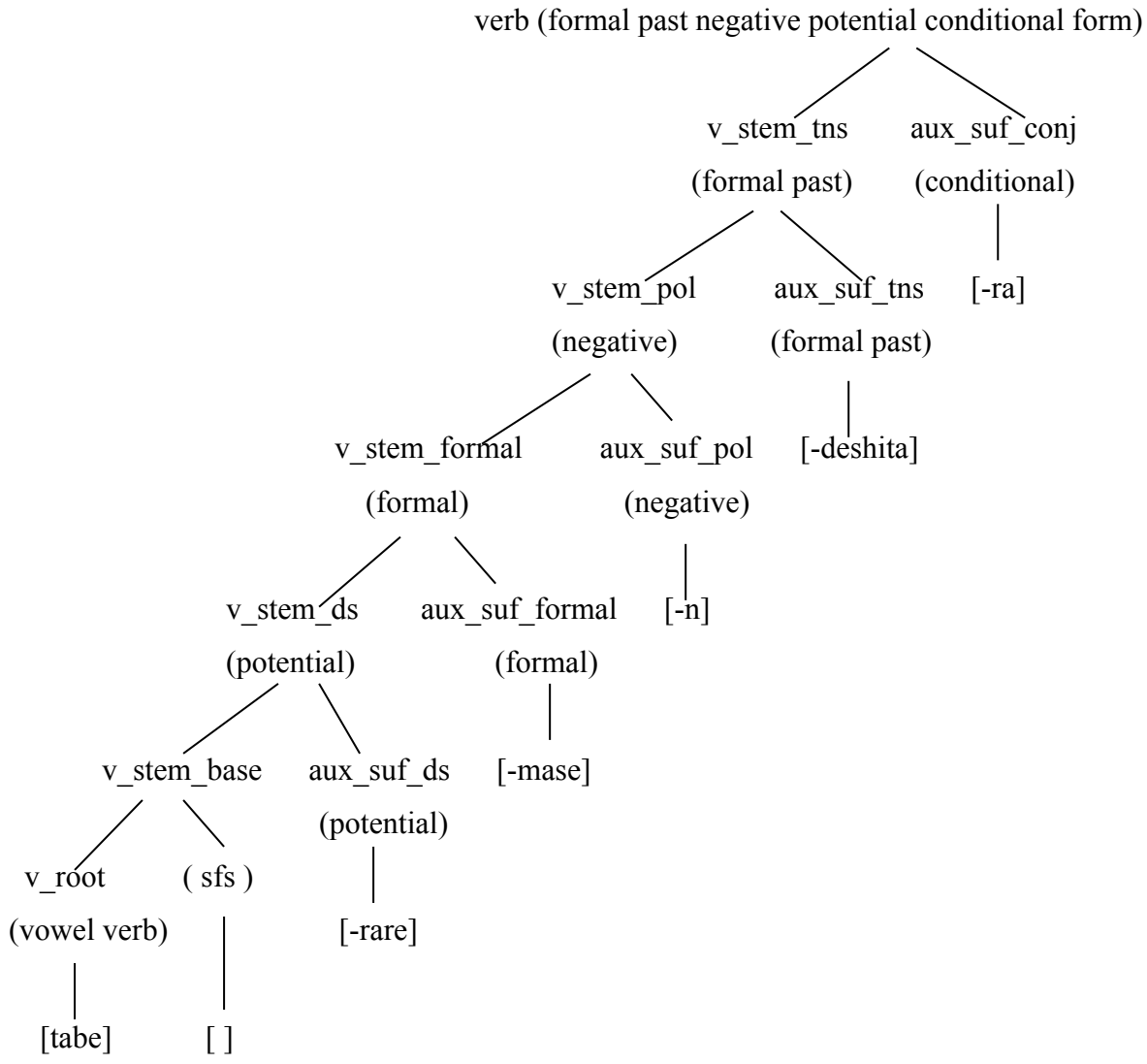
→ [Tense]	**→ [Conjunctive Suffix]**
-*ru/-u* (Present)	-*ra/-reba/-ba* (Conditional)
-*i* (Present negative)	-*te*/-de (Continuative)
-*ta/-da* (Past)	-*o/-yoo* (Volitional)
-*katta* (Past negative)	-[]/-*ro* (Imperative)
-*deshita* (Formal past negative/negative conditional)	-[] (Other forms)
-*runa* (Present negative imperative)	
-[] (Other forms)	

Based on this auxiliary suffix sequence analysis, the formal past negative potential conditional form, '*tabe rare mase n deshita ra*' meaning 'if (I) not able to eat' is constructed as shown below:

[*tabe*]	-[*rare*]	-[*mase*]	-[*n*]	-[*deshita*]	-[*ra*]
[**Stem**]	**-Derivational Suffix**	**-Formality**	**-Polarity**	**-Tense**	**-Conjunction**
[eat]	-[potential]	-[formal]	-[negative]	-[past formal]	-[conditional]

Auxiliary suffixes appear to form a linear sequencial strucutre. However, in the word formation analysis discussed in Section 2, the various auxiliary suffixes attach to a verb stem recursively to form a verb. Tree 9 shows the recursive tree sturucture of the verb form with various auxiliary suffixes attached to construct the above verb form of formal past negative potential conditional, **'*tabe rare mase n deshita ra*'** by applying the word formation rules defined in Section 2.

[Tree 9] Tree Structure of Formal Past Negative Potential Conditional Form of the Verb *"taberu"*



This structure allows complex auxiliary suffixes to generate successfully their own conjugation and formation recursively to construct a Japanese verb form.

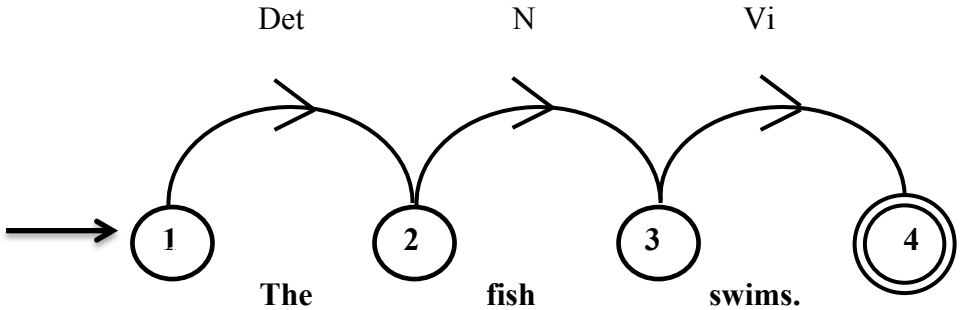
6. PROLOG PROGRAMMING

6.1 What Is Prolog?

Prolog is a logic programming language and its name comes from the short for "**pro**gramming in **log**ic." As it says in the name, it is suitably used to solve problems logically by involving the relations between objects and objects. Matthews (1998:33) states the trait of relation-based Prolog language that "[a] prolog program consists of a number of statements defining various relationships which hold between the objects in the area of interest." Prolog is suitable for natural language analysis because language is rule-based and each element in language structure is related as I mentioned in discussing word formation rules in Section 2. Pereira and Shieber (1987:2) address the relation between Prolog computer language and natural language analysis, "[T]he development of logic programming has been closely tied to the search for computational formalisms for expressing syntactic and semantic analyses of natural-language sentences."

6.2 Natural Language Analysis for Prolog Programming

In Prolog, there are two alternative grammar representations for programming: Finite-State Grammar and Phrase Structure Grammar. Naturally, we think a sentence structure can be represented as a string of words. Matthews defines sentence structure in Finite-State Grammar as "a graph [that] consists of a set of nodes linked together by arcs," and shows its graphic figure with the example sentence "The fish swims" as shown below:



(Matthews, 1998:128-129)

Figure 1: The Example Sentence Structure "The fish swims."

Figure 1 consists of three components: determiner, noun, and verb; and they connect to each other with arrow arcs to show the sentence relation as the string of words of Det N Vi. This graphic representation is called "Finite-State Transition Network (FSTN)" and the resulting

grammar is called “Finite-State Grammar.” This representation makes it look comparatively simple to characterize a string of words in a sentence structure as a linear sequence; however, the word formation rules stated in Section 2, construct a binary tree structure to define lexicon formations. Therefore, Matthews (1998:160) introduces “Context-Free Grammar (CFG)” as an alternative analysis of FSTN.

> The expressive power of Prolog means that it is relatively easy to define recognition procedures based on FSTNs … However, more concise logical formalisms are possible which are better suited for expression in Prolog [which is context-free grammar].

While Finite-State Grammar defines a procedural formalism, Context-Free Grammar is declarative with structured rules. Pereira and Shieber (2002:22) also identify CFG as a beneficial structure analysis for rule-based word formation: “*Context-free grammars* (CFG) constitute a system for defining the expressions of a language in terms of *rules.*” CFG is suitable for applying the process-neutral statement of word formation to Prolog programming. Matthews (1998:164-165) reinforces the advantages of CFG over FSTN system as follows:

> [In terms of] a declarative description of *what* is being computed (a grammar) and a specification of *how* the actual computations may be produced (a recognition or parsing procedure) … [FSTN] often blur the distinction between *what* and *how*… Prolog is ideally suited to the task of representing phrase structure grammars.

For these reasons, I follow the system of “Context-Free Grammar” in order to apply the rule-based word formation analysis of Japanese verb conjugation to Prolog programming.

6.3 Prolog Program in General Example

Prolog deals with the relation of objects logically by defining rules. Bratko (1986:3-8) uses a family relation to explain the basic mechanisms of Prolog. The family tree representation is simplified in Figure 2 in order to explain the simple grandparent relation in this section.

Figure 2: Simple Grandparent Relation

Figure 2 expresses the family relation between Tom, Bob, and Ann: Tom is a parent of Bob and Ann is a parent of Sally. This family relation can be represented as rules in the Prolog program as follows:

parent(tom, bob).
parent(bob, ann). (Bratko, 1986:3-8)

In Prolog logic, **parent** is used to name the relation as "predicate;" and **tom**, **bob** and **ann** are its "arguments." All of the written words as of "predicate" and "arguments" have to be typed in lower-case letters. Each of the rules is defined as "clause," which states one fact about the parent relation and has to be ended with a period. A Prolog program is constructed with a number of "clauses" to define the rule relations and solve problems related to the relation. Figure 2 consists of two clauses to define the parent relation of Tom, Bob, and Ann.

The prolog program can communicate by responding to the questions about the parent relation such as "Is Tom a parent of Bob?" The question clause below is called **query** in Prolog.

?- parent(tom, bob).

When we input the question in query, Prolog tries to find the answer by matching the same fact as stated in the program. The question **parent (tom, bob)** matches the first clause in the program: **parent(tom, bob).** so the program will answer:

yes

But when we ask "Is Tom a parent of Ann?" with the query below:

?- parent(tom, ann).

Prolog answers:

no

This is because the program will not find the identical clause to the statement in the query. Also, if someone, who is not identical in the clauses, is defined in the query below:

?- parent(tom, peter).

Prolog answers:

no

This is because the program will not find the name of **peter** in the clauses so the program denies the query.

Furthermore, we can ask the question "Who is the parent of Bob?" to the program. In a Prolog program, any alphabetical letters or words, such as X or Who, can be used to identify "who" in the query. However, the letter or the initial of the word has to be capitalized in a query because those letters or words will be replaced with applicable words in the result. This alternative argument is called **variable**. The below query uses "Who" as a variable to ask the question "Who is the parent of Bob?"

?- parent(Who, bob).

Then the answer is:

Who = tom

First, the program tries to match the identical clause in the program. When it finds the identical clause, it finds the word that fits in the position to match the clause.

With this query, we can ask who the child of Bob:

?- parent(bob,Who).

Prolog answers:

Who = ann

Also, it is possible to ask a more extended question such as "Who is the parent of whom?"

?- parent(Who,Whom).

Then the program will find the all possible parent-child relations and display all solutions with a semi-colon ';' meaning "or." After the program displays all the solutions, it finishes with "No more solutions" as shown below.

Who = tom

Whom = bob;

Who = bob

Whom=ann;

No more solutions.

Moreover, the grandparent relation can also be defined in the Prolog program. This extended relation requires two steps to be defined in order to construct queries:

1. Who is a parent of Ann?
2. Who is a parent of Ann's parent?

In this query, the letters, **X** and **Y**, can be used to identify Ann's parent as **X** and the parent of Ann's parent as **Y** to avoid the complication of the use of interrogative pronouns, **who** and **whom**. With the classification **X** and **Y**, the two sequences of queries can be stated with a comma ',' meaning "and" as follows:

?- parent(X, ann), parent(Y, X).

Because the query consists of two facts of the family relation combining with the comma ',' in a single clause, it requires to satisfy with two fact relations to identify **X** and **Y**. So, the answer is:

X = bob

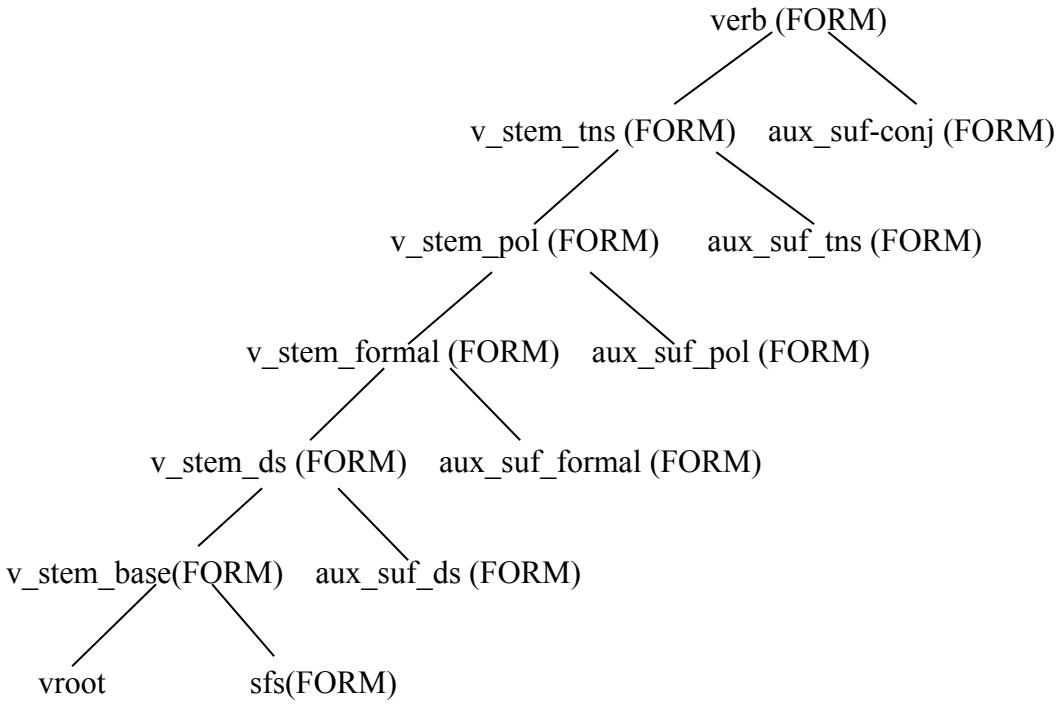
Y = tom

This is a basic mechanism of matching and identifying the relation clauses to solve problems in Prolog. In sum, Prolog can easily define a relation consisting of clauses with a predicate and its arguments. The argument can be a constant or a variable. Arguments can be persons (such as Tom, Sam, and Sally), numbers, interrogative pronouns (such as Who or Whom) or general substitute letters such as X and Y. Bratko (1986:7) provides further definition of arguments, stating that the concrete objects such as persons and numbers written in lower-case letters are called "atoms" and the undefined objects written in capitalized letters such as Who, Whom, X, and Y are called "variables."

6.4 Japanese Verb Conjugation in Prolog

In this section I build up the predicate logic rules of Japanese verb conjugation for a Prolog programming. As I mentioned in Section 2.3, Japanese verb conjugation can be represented in a tree structure with its word formation rules. The word formation rule of Japanese verb conjugation is defined in Tree 10.

[Tree 10] Word Formation Rule of Japanese Verb Conjugation



According to Tree 10, Japanese verb conjugation is described with rules in Prolog programing as follows:

verb --> v_stem_tns(FORM), aux_suf_conj(FORM).
v_stem_tns(FORM) --> v_stem_pol(FORM), aux_suff_tns(FORM).
v_stem_pol(FORM) --> v_stem_formal(FORM), aux_suff_pol(FORM).
v_stem_formal(FORM) --> v_stem_ds(FORM), aux_suff_formal(FORM).
v_stem_ds(FORM) --> v_stem_base(FORM), aux_suff_ds(FORM).
v_stem_base(FORM) --> vroot, sfs(FORM).

The Prolog rules for the two verb entries, *"kaku"* and *"taberu,"* with two forms, the imperative and causative forms, such as 'kak-a-na-i' and 'tabe-na-i' as the imperative form and 'kak-a-se-ru' and 'tabe-sase-ru' as the causative form, are given in the Prolog programing below:

(1) %%%Japanese verb form1.pl

```
verb(FORM) --> v_stem_tns(FORM), aux_suf_conj(FORM).
v_stem_tns(FORM) --> v_stem_pol(FORM), aux_suf_tns(FORM).
v_stem_pol(FORM) --> v_stem_formal(FORM), aux_suf_pol(FORM).
v_stem_formal(FORM) --> v_stem_ds(FORM), aux_suf_formal(FORM).
```

```
v_stem_ds(FORM) --> v_stem_base(FORM), aux_suf_ds(FORM).
v_stem_base(FORM) --> vroot, sfs(FORM).

vroot --> [kak].
vroot --> [tabe].

sfs(imperfective) --> [-a].
sfs(causative) --> [-a].

aux_suf_ds(imperfective)  --> [].
aux_suf_ds(causative)  --> [-sase];[-se].

aux_suf_formal(imperfective)  --> [].
aux_suf_formal(causative)  --> [].

aux_suf_pol(imperfective)  --> [-na].
aux_suf_pol(causative)  --> [].

aux_suf_tns(imperfective)  --> [-i].
aux_suf_tns(causative)  --> [-ru].

aux_suf_conj(imperfective)  --> [].
aux_suf_conj(causative)  --> [].
```

The new notations in the above program are **'%'** and **'- - >.'** **'%'** is used to express comments which are not a part of the program such as "title of the program." And **'- - >'** is used in the predicate logic as to mean "may consist of" in a word formation rule. And the entries in the Prolog program such as verb roots, SFS and an auxiliary suffixes are displayed with the bracket **'[ ].'** The apparent problem in Program (1) is that it does not clarify which SFS is attached to which verb roots and which auxiliary suffixes are followed by which verb stems. Therefore, I add some classifications regarding the verb group on each predicate. Program (2) below defines Group 'cs' for consonant verbs and Group 'vw' for vowel verbs as well as main verb entry (MVE).

(2) %%%Japanese verb form2.pl MVE: Main Verb Entry

```
verb(MVE,GROUP,FORM)-->v_stem_tns(MVE,GROUP,FORM), aux_suf_conj(FORM).
v_stem_tns(MVE,GROUP,FORM)-->v_stem_pol(MVE,GROUP,FORM), aux_suf_tns(FORM).
```

```
v_stem_pol(MVE,GROUP,FORM)-->v_stem_formal(MVE,GROUP,FORM),
aux_suf_pol(FORM).
v_stem_formal(MVE,GROUP,FORM)-->v_stem_ds(MVE,GROUP,FORM),
aux_suf_formal(FORM).
v_stem_ds(MVE,GROUP,FORM)-->v_stem_base(MVE,GROUP,FORM), aux_suf_ds(FORM).
v_stem_base(MVE,GROUP,FORM)-->vroot(MVE,GROUP), sfs(FORM).

vroot(kaku,cs)  --> [kak].
vroot(taberu,vw)  --> [tabe].

sfs(imperfective) --> [-a].
sfs(causative) --> [-a].

aux_suf_ds(imperfective)  --> [].
aux_suf_ds(causative)  --> [-sase];[-se].

aux_suf_formal(imperfective)  --> [].
aux_suf_formal(causative)  --> [].

aux_suf_pol(imperfective)  --> [-na].
aux_suf_pol(causative)  --> [].

aux_suf_tns(imperfective)  --> [-i].
aux_suf_tns(causative)  --> [-ru].

aux_suf_conj(imperfective)  --> [].
aux_suf_conj(causative)  --> [].
```

Program (2) elicits the imperative and causative forms of the verbs *"kaku"* and *"taberu."* Like the family relation, the predicate rules include some variables such as groups and forms. These variables are matched with each categorized entry in a verb root, SFS, and an auxiliary suffix to execute a query. Therefore, in the query asking for all verb forms in the program, all variable arguments including VERB itself need to be stated in a parenthesis **'( )'** with an empty bracket [ ] as shown:

verb(MVE,GROUP,FORM,VERB,[]).

Then, the solutions or answers of the query will be:

```
[1] ?-verb(MVE,GROUP,FORM,VERB,[]).
Soln: 1
MVE = kaku; GROUP = cs; FORM = imperfective; VERB = [kak, - a, - na, - i]
;
Soln: 2
MVE = kaku; GROUP = cs; FORM = causative; VERB = *[kak, - a, - sase, - ru]
;
Soln: 3
MVE = kaku; GROUP = cs; FORM = causative; VERB = [kak, - a, - se, - ru]
;
Soln: 4
MVE = taberu; GROUP = vw; FORM = imperfective; VERB = *[tabe, - a, - na, - i]
;
Soln: 5
MVE = taberu; GROUP = vw; FORM = causative; VERB = *[tabe, - a, - sase, - ru]
;
Soln: 6
MVE = taberu; GROUP = vw; FORM = causative; VERB = *[tabe, - a, - se, - ru]
;
No More Solutions.
```

In the above solutions, some mistakes of the verb conjugation exist as indicated above with asterisks. There are two auxiliary suffixes in causative form depending on verb groups. But the above predicate logic rules do not define the classification of auxiliary suffixes following to each verb group. This misclassification results in the duplication of auxiliary suffixation to elicit the wrong verb formations. For example, in the above results, the Prolog program elicits two causative forms of the verb *"kaku,"* [kak, - a, - se, - ru] and *[kak, - a, - sase, - ru]. However, the latter form *[kak, - a, - sase, - ru] is an incorrect form derived from the auxiliary suffixes duplication. The solution for these overlapped derivations is to specify the form of auxiliary suffixes, such as "causative1" for vowel verbs and "causative2" for consonant verbs to distinguish between these two auxiliary suffixes. These form classifications are required only for certain forms of auxiliary suffixes. Thus, in Program (2), the auxiliary suffix in the imperfective form does not need to be defined, but the suffixes in the causative and passive forms need to be specified by each group. The suffix rules are modified as below:

```
aux_suf_ds(imperfective)  --> [].
aux_suf_ds(causative1)  --> [-sase].
aux_suf_ds(causative2)  --> [-se].

aux_suf_formal(imperfective)  --> [].
aux_suf_formal(causative1)  --> [].
aux_suf_formal(causative2)  --> [].

aux_suf_pol(imperfective)  --> [-na].
aux_suf_pol(causative1)  --> [].
aux_suf_pol(causative2)  --> [].

aux_suf_tns(imperfective)  --> [-i].
aux_suf_tns(causative1)  --> [-ru].
aux_suf_tns(causative2)  --> [-ru].

aux_suf_conj(imperfective)  --> [].
aux_suf_conj(causative1)  --> [].
aux_suf_conj(causative2)  --> [].
```

Another problem with the Prolog program is unnecessary SFS attachments on vowel verbs. SFS is attached to only consonant verbs not to vowel verbs; however, the program derives all vowel verb *"taberu"* with SFS attached. In order to avoid unnecessary SFS attachment on vowel verbs, one more distinct variable, GROUP, is added in to specify which group of verbs require SFS. Also, the form in SFS needs to follow the same form classification as that of the auxiliary suffixes to derive the correct form of the verb. The revised predicate logic rules and SFS clauses is as follows:

```
v_stem_base(MVE,GROUP,FORM) --> vroot(MVE,GROUP), sfs(FORM,GROUP).

sfs(imperfective,cs) --> [-a].
sfs(imperfective,vw) --> [].
sfs(causative2,cs) --> [-a].
sfs(causative1,vw) --> [].
```

The program seems to be fixed with these specific classifications of grouping SFS and auxiliary suffixes to elicit all verb forms of imperfective and causative with consonant verb of *"kaku"* and vowel verb of *"taberu,"* correspondingly. However, one more problem needs to be solved to complete the program. That is the specific breakdown of imperfective forms. There are five semantically different imperfective forms of auxiliary suffixes, which need to be included in the program: **[-na-i]; [-nakat-ta]; [-naku-te]; [-nake-reba]; [-nakat-ta-ra].** All of them carry slightly different semantic meanings regarding tense, polarity, and formality. Therefore, I add these features: tense, polarity, and formality, in the FORM variable: **FORM/TENSE/POLARITY/FORMALITY** in order to classify each different functional form of auxiliary suffixes. The slash in the FORM variable is an efficient technique in Prolog Programming to break down various features in a single variable and it allows each feature to function independently within the variable. According to Sobin (2013:71), the function of the slash notation is "to divide a single argument position into discrete parts, any of which may be either a constant or a variable." In addition, I include English translation "**TRANS**" as the variable in the arguments for each verb root entry. The complete modified Prolog program is shown in Program (3).

(3) %%%Japanese verb form3.pl

```
verb(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_tns(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_conj(FORM/TENSE/POLARITY/FORMALITY).

v_stem_tns(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_pol(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_tns(FORM/TENSE/POLARITY/FORMALITY).

v_stem_pol(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_formal(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_pol(FORM/TENSE/POLARITY/FORMALITY).

v_stem_formal(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_ds(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_formal(FORM/TENSE/POLARITY/FORMALITY).

v_stem_ds(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_base(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
```

```
aux_suf_ds(FORM/TENSE/POLARITY/FORMALITY).

v_stem_base(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
vroot(MVE,TRAN,GROUP), sfs(FORM,GROUP).

vroot(kaku,to_write,cs)  --> [kak].
vroot(taberu,to_eat,vw)  --> [tabe].

sfs(imperfective,cs) --> [-a].
sfs(imperfective,vw) --> [].
sfs(causative2,cs) --> [-a].
sfs(causative1,vw) --> [].

aux_suf_ds(imperfective/present/negative/informal)  --> [].
aux_suf_ds(imperfective/past/negative/informal)  --> [].
aux_suf_ds(imperfective/present/negative_continuative/informal)  --> [].
aux_suf_ds(imperfective/present/negative_conditional/informal)  --> [].
aux_suf_ds(imperfective/past/negative_conditional/informal)  --> [].
aux_suf_ds(causative1/present/affirmative/informal)  --> [-sase].
aux_suf_ds(causative2/present/affirmative/informal)  --> [-se].

aux_suf_formal(imperfective/present/negative/informal)  --> [].
aux_suf_formal(imperfective/past/negative/informal)  --> [].
aux_suf_formal(imperfective/present/negative_continuative/informal)  -->
[].
aux_suf_formal(imperfective/present/negative_conditional/informal)  --> [].
aux_suf_formal(imperfective/past/negative_conditional/informal)  --> [].
aux_suf_formal(causative1/present/affirmative/informal)  --> [].
aux_suf_formal(causative2/present/affirmative/informal)  --> [].

aux_suf_pol(imperfective/present/negative/informal)  --> [-na].
aux_suf_pol(imperfective/past/negative/informal)  --> [-nakat].
aux_suf_pol(imperfective/present/negative_continuative/informal)  -->
[-naku].
aux_suf_pol(imperfective/present/negative_conditional/informal)  -->
[-nake].
aux_suf_pol(imperfective/past/negative_conditional/informal)  --> [-nakat].
aux_suf_pol(causative1/present/affirmative/informal)  --> [].
```

```
aux_suf_pol(causative2/present/affirmative/informal)  --> [].

aux_suf_tns(imperfective/present/negative/informal)  --> [-i].
aux_suf_tns(imperfective/past/negative/informal)  --> [-ta].
aux_suf_tns(imperfective/present/negative_continuative/informal)  --> [].
aux_suf_tns(imperfective/present/negative_conditional/informal)  --> [].
aux_suf_tns(imperfective/past/negative_conditional/informal)  --> [-ta].
aux_suf_tns(causative1/present/affirmative/informal)  --> [-ru].
aux_suf_tns(causative2/present/affirmative/informal)  --> [-ru].

aux_suf_conj(imperfective/present/negative/informal)  --> [].
aux_suf_conj(imperfective/past/negative/informal)  --> [].
aux_suf_conj(imperfective/present/negative_continuative/informal)  -->
[-te].
aux_suf_conj(imperfective/present/negative_conditional/informal)  -->
[-reba].
aux_suf_conj(imperfective/past/negative_conditional/informal)  --> [-ra].
aux_suf_conj(causative1/present/affirmative/informal)  --> [].
aux_suf_conj(causative2/present/affirmative/informal)  --> [].
```

In order to add each argument to each verb root entry in a program, Pereira and Shieber (2002:59) introduce the encoding lexicon entry technique in Prolog programming: "[o]ne of the most common uses for adding Prolog goals to [a DCG] is the simplification of the encoding of the lexicon:"

n - - > [Word], {n(Word)}.

According to the encoding lexicon entry technique by Pereira and Shieber, the two verb root entries in Program (3) can be defined as follows.

```
vroot(MVE,TRANS,GROUP) --> [Vroot],{vroot(Vroot,MVE,TRANS,GROUP)}.
       vroot(kak,'kaku',to_write,cs).
       vroot(tabe,'taberu',to_eat,vw).
```

With this programming technique, I include the 306 verb roots in the program.

6.5 Consonant Verbs and *Onbin* Process in Prolog

Lastly, I explain how the typical Japanese morphophonemic process '*Onbin*' is

included in the Prolog programming. Most of the *Onbin* processes occur in consonant verbs when they attach to the auxiliary suffixes in the continuative and the past-tense forms followed by *-te* and *-ta*; and they are classified into three *Onbin* processes mentioned in Section 4. The following is a brief summary of three *Onbin* processes:

Process 1 occurs when the root of consonant verbs ends in *-t, -r, -u(w)* and they are changed to /t/ through assimilation process. Process 1 also involves three phoneme alternations: consonant insertion of /w/ between the vowel sequence; the combination of palatalization and affrication which change the phoneme from /t/ to /tš/ as 'ch'; and affrication which changes the phoneme from /t/ to /ts/ as 'ts.'

Process 2 involves two phoneme alternations. Nasalization occurs when the root of consonant verbs ends in *-n, -m, -b* changing the phoneme to /n/. After the root end has been nasalized, the process of assimilation changes the phoneme /t/ to /d/ in the continuative and past-tense forms.

Process 3 occurs when the phoneme /i/ attaches to the consonant verbs ending in the roots *-k* and *-g*. Palatalization occurs to change the roots *k* and *g* ending consonants to k^y and g^y. Afterwards, both of the palatalized consonants are assimilated to become /i/ by vowelization. Process 3 also includes two abstract phonemes /s/ and /š/ representations in Japanese syllabary writing system.

In order to execute these *Onbin* processes in a Prolog program, I address two root variations for all consonant verbs with distinguishable matching group codes: group classification 'cs' for regular roots of consonant verbs, 'cs/process1' for the alternatives in Process 1 and 'cs/process3' for the alternatives in Process 3, and 'cs/process2/de/da' for the alternative to define the Process 2 alternation of /t/ to /d/. The below clauses show the example verb root entries of each *Onbin* process in the program.

```
%%% Consonant verbs: cs
%%% Consonant verbs -Onbin Process 1
vroot(agar,'agaru',to_go_up,cs).
vroot(agat,'agaru',to_go_up,cs/process1).

%%% Consonant verbs -Onbin Process 2
vroot(asob,'asobu',to_play,cs).
```

```
vroot(ason,'asobu',to_play,cs/process2/de/da).

%%% Consonant verbs -Onbin Process 3
vroot(ak,'aku',to_open/intrans,cs).
vroot(ai,'aku',to_open/intrans,cs/process3).
```

As for the other phonemic alternations, I define the specific codes for each classification. The verbs in Process 1 that include /w/ insertion are encoded as 'cs/irreg1/wa' to distinguish from the entry of *Onbin* process 1. The below is the example of *"arasou"* meaning "to fight."

```
%%% Consonant verbs -Onbin Process 1
vroot(araso,'arasou',to_fight,cs/irreg1/wa).
vroot(arasot,'arasou',to_fight,cs/process1).
```

The verbs involving palatalization and affrication of /ch/ and affrication of /ts/ in Process 1 are encoded in four different groups as 'cs/t,' 'cs/ts,' 'cs/ch,' and 'cs/process/1,' respectively. The below is the example verb root entry of *"katsu."*

```
%%% Consonant verbs -Onbin Process 1
vroot(kat,'katsu',to_win,cs/t). vroot(kats,'katsu',to_win,cs/ts).
vroot(kach,'katsu',to_win,cs/ch). vroot(kat,'katsu',to_win,cs/process1).
```

In Process 3, with respect to the /s/ and /sh/ alternation, the codes 'cs/s' and 'cs/irreg3/sh' are used to distinguish between the verb root with /s/ ending and the verb root with /sh/ ending. The below is the example of *"arawasu"* meaning "to appear."

```
%%% Consonant verbs -Onbin Process 3
vroot(arawas,'arawasu',to_appear/trans,cs/s).
vroot(arawash,'arawasu',to_appear/trans,cs/irreg3/sh).
```

Additionally, there are two consonant verbs in Process 1 which require more than two root variations irregularly: *"aru"* meaning "to be" and *"irassharu"* meaning "to go, come or be in honorific expression." Each of them requires three root variations: 'ar,' '[],' and 'at' for *"aru"* and 'irasshar,' 'irassha,' and 'irasshat' for *"irassharu."* The distinguishable codes of each verb root variation in the verb root clauses are represented as follows:

```
vroot(ar,'aru',to_be,cs/ar).  vroot([],'aru',to_be,cs/[]).
vroot(at,'aru',to_be,cs/process1).

vroot(irasshar,'irassharu',to_go/come/be/honorific,cs/irasshar).
vroot(irassha,'irassharu',to_go/come/be/honorific,cs/irreg2/irassha).
vroot(irasshat,'irassharu',to_go/come/be/honorific,cs/process1).
```

In the *Onbin* process, each group code of SFS must match the code of the root's group in order to derive the applicable SFS to be attached to the verb root. Therefore, I include the various group codes in SFS clauses in the program.

Example SFS for consonant verbs in the imperfective form:

```
sfs(imperfective,cs) --> [-a].                : For regular roots
sfs(imperfective,cs/irreg1/wa) --> [-wa].: For verb roots with /w/ insertion in Onbin
                                               Process 1
sfs(imperfective,cs/s) --> [-a].              : For /s/ ending verb roots in Onbin Process
                                               3
sfs(imperfective,cs/t) --> [-a].              : For verb roots with /t/ affrication in Onbin
                                               Process 1
sfs(imperfective,cs/irasshar) --> [-a]. : For the irregular consonant verb
                                               "irrasharu"
sfs(imperfective,cs/[]) --> [].               : For the irregular consonant verb "aru"
```

Example SFS for consonant verbs in the polite form:

```
sfs(polite,cs) --> [-i].                      : For regular roots
sfs(polite,cs/irreg1/wa) --> [-i].            : For verb roots with /w/ insertion in Onbin
                                                Process 1
sfs(polite,cs/irreg3/sh) --> [-i].            : For /sh/ ending verb roots in Onbin
                                                Process 3
sfs(polite,cs/ch) --> [-i].                   : For verb roots with /ch/ palatalization and
                                                affrication in Onbin Process 1
sfs(polite,cs/ar) --> [-i].                   : For the irregular consonant verb "aru"
sfs(polite,cs/irreg2/irassha) --> [-i]. : For the irregular consonant verb
                                                "irrasharu"
```

Example SFS for consonant verbs in the continuative form:

```
sfs(continuative1,cs/process1) --> [].
```
: For the verb roots in the *Onbin* Process 1

```
sfs(continuative1,cs/process3) --> [].
```
: For the verb roots in the *Onbin* Process 3

```
sfs(continuative2,cs/process2/de/da) --> [].
```
: For the verb roots in the *Onbin* Process 2.

```
sfs(continuative1,cs/irreg3/sh) --> [-i].
```
: For /sh/ ending verb roots in *Onbin* Process 3.

6.6 Vowel Verbs and Two Irregular Verbs in Prolog

Compared with consonant verbs, vowel verbs simply follow the word formation rule without any irregularities. The verb entry is simply one root for each verb. The example verb root entries of *"ageru"* meaning "to raise" and *"akeru"* meaning "to open" are as follows.

```
vroot(age,'ageru',to_raise,vw).
vroot(ake,'akeru',to_open/trans,vw).
```

Vowel verbs belonging to Group 'vw' do not require SFS to form the verb stem. So, SFS entry for a vowel verb is just blank '[].'

```
sfs(imperfective,vw) --> [].
sfs(causative1,vw) --> [].
sfs(passive1,vw) --> [].
sfs(polite,vw) --> [].
sfs(continuative1,vw) --> [].
sfs(past_tense1,vw) --> [].
sfs(conditional_past1,vw) --> [].
sfs(predicative2,vw) --> [].
sfs(conditional2,vw) --> [].
sfs(imperative2,vw) --> [].
sfs(negative_imperative2,vw) --> [].
sfs(volitional2,vw) --> [].
sfs(potential2,vw) --> [].
sfs(potential3,vw) --> [].
```

On the other hand, the two irregular verbs in Group 'irreg', kuru-irregular verb and

suru-irregular verb, share both characteristics of a consonant verb and a vowel verb. One characteristic that they share with a consonant verb is they carry more than one verb root variation to conjugate. kuru-irregular verb has three root variations: *'ko,' 'ki,'* and *'ku;'* and suru-irregular verb has four variations: *'sa,' 'shi,' 'su,'* and *'de,'* respectively. The verb root entries for these two verbs are shown below.

```
%%% "kuru" irregular verb: irreg
vroot(ko,'kuru',to_come,irreg/ko). vroot(ki,'kuru',to_come,irreg/ki).
vroot(ku,'kuru',to_come,irreg/ku).

%%% "suru" irregular verb: irreg
vroot(sa,'suru',to_do,irreg/sa).    vroot(shi,'suru',to_do,irreg/sh).
vroot(su,'suru',to_do,irreg/su).     vroot(de,'suru',to_do,irreg/de).
```

The other characteristic that the two irregular verbs share with a vowel verb is that they do not require SFS attachments, so the entries of SFS for these irregular verbs are all blank '[].'

```
sfs(imperfective,irreg/ko) --> [].
sfs(imperfective,irreg/sh) --> [].
sfs(causative1,irreg/ko) --> [].
sfs(causative2,irreg/sa) --> [].
sfs(passive1,irreg/ko) --> [].
sfs(passive2,irreg/sa) --> [].
sfs(polite,irreg/ki) --> [].
sfs(polite,irreg/sh) --> [].
sfs(continuative1,irreg/ki) --> [].
sfs(continuative1,irreg/sh) --> [].
sfs(past_tense1,irreg/ki) --> [].
sfs(past_tense1,irreg/sh) --> [].
sfs(conditional_past1,irreg/ki) --> [].
sfs(conditional_past1,irreg/sh) --> [].
sfs(predicative2,irreg/ku) --> [].
sfs(predicative2,irreg/su) --> [].
sfs(conditional2,irreg/ku) --> [].
sfs(conditional2,irreg/su) --> [].
sfs(imperative3,irreg/ko) --> [].
sfs(imperative2,irreg/sh) --> [].
```

```
sfs(negative_imperative2,irreg/ku) --> [].
sfs(negative_imperative2,irreg/su) --> [].
sfs(volitional2,irreg/ko) --> [].
sfs(volitional2,irreg/sh) --> [].
sfs(potential2,irreg/ko) --> [].
sfs(potential4,irreg/de) --> [].
```

Included all these specific features, the finalized version of the Prolog program is shown in Appendix 1.

6.7 The Process of Eliciting Japanese Verb Conjugation in a Proof Tree

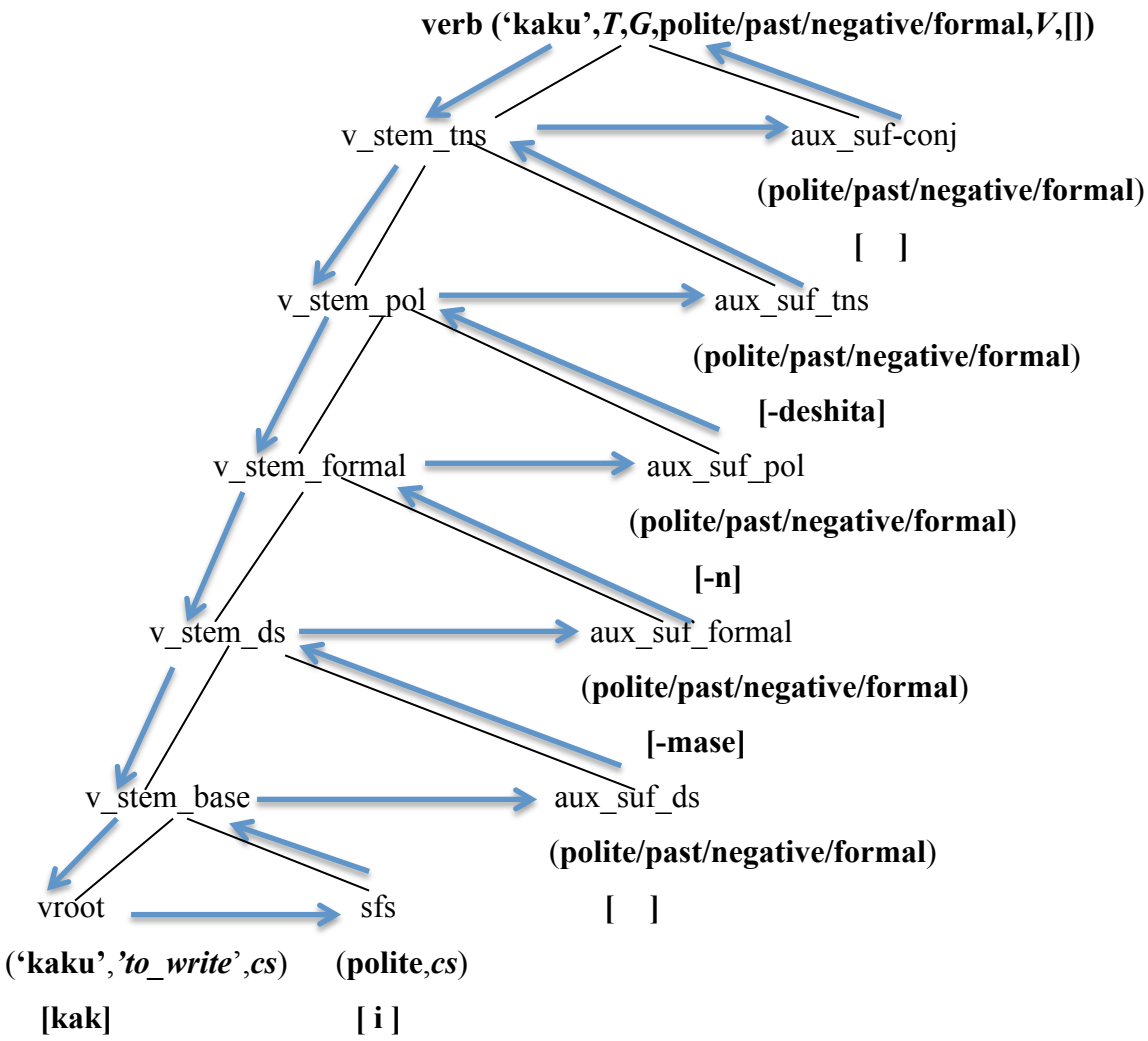
A tree structure can be used to explain how Prolog derives a word. The tree structure, showing word formation, is called 'proof tree' in Prolog. In terms of Prolog programming, Pereira and Shieber (2002:15) explain the "proof tree" stating: "[this proof tree] makes explicit the steps in the execution of Prolog that led to a successful proof of the goal under a given assignment." In a recognition process in a proof tree, although there is a variety of ways to access and manipulate the word formation rules, it is usually operates as top-down or bottom-up, left-to-right or right-to-left procedure based on what a query seeks. Matthews (1998:167) explains the distinction of these procedures by using a phrase structure rule, the rule *S - -> NP VP*. A top-down procedure will access this rule initially from its left-hand side to seek the goal S. Once the mother category has been identified, the information to the right of the arrow becomes available, followed by the procedure to seek only NP first, and when that's resolved, VP. On the contrary, a bottom-up procedure would start from the lower position, resolving NP first, and then VP, before establishing S. Tree 11 below represents the top-down left-to-right parsing process to elicit the Japanese verb "*kaku*" in the formal past negative form, '*kak-i-mase-n-deshita*.' The Prolog process begins from the query, verb('kaku',Translation,Group,polite/past/negative/formal,Verb,[])., and the goal/output of the execution is to elicit the appricable features matching with each variable in the argument of the query, which are Translation=to_write, Group=cs, Verb=[kak,-i, -mase,-n,-deshita].

[Tree 11] Proof Tree Structure of Formal Past Negative Form of the Verb *"kaku"*

Query: verb('kaku',T,G,polite/past/negative/formal,V,[]).

T: Translation, G: Group, V: Verb

Output: T = to_write, G = cs, V = [kak,-i,-mase,-n,-deshita]



In Tree 11, the execution begins with the query, verb('kaku',Translation,Group,polite/past/negative/formal,Verb,[]). The goal of the execution is to value each variable (variables here begin with an upper-case letter) in the query. The variables here are Translation, Group, and Verb. In order to accomplish the goal, a top-down left-to-right tracking is used to assign the corresponding features to the variables in the query based on the feature information of the defined main verb entry and form. In a top-down parsing process, the main verb entry "kaku" and the form "polite/past/negative/formal" in the query are passed on from the highest phrasal level down the left side to the bottom element 'verb root.' The main verb entry

information is used to fill up all required variables, Translation and Group, of the verb root. Then, the form and verb group classification project the variables of SFS to determine the required stem forming suffix in the formal past negative form of "kaku." After assigning the required features in 'verb root' and 'sfs' to elicit a part of the target verb form, [kak] and [i], the parsing process exits in "v_stem_ds." Then, it starts to call continuously each auxiliary suffix to project the target auxiliary suffixes, [-mase], [-n], and [-deshita], by matching to the form feature "polite/past/negative/formal." This tracking process executes the matching features to achieve the goal of finding "kak, -i, -mase, -n, -deshita" as the formal past negative form of *"kaku."* This is how a proof tree shows how all variables are valued with the appropriate features to elicit the target solution (a correct Japanese verb form) successfully. In the next section, I analyze the various outputs from the program.

7. DISCUSSION OF THE OUTPUTS IN JAPANESE VERB CONJUGATION IN PROLOG PROGRAM

In this section, I demonstrate the Prolog program of Japanese verb conjugation and analyze the various outputs of the program. This program includes a total 306 Japanese verbs (304 regular and 2 irregular verbs) and can derive 23 various verb forms of consonant verbs and two irregular verbs and 24 various verb forms of vowel verbs. The program can provide various Japanese verb forms in various types of query. I will demonstrate eight types of the queries with respect to verb forms, verb stems, verb roots, auxiliary suffixes, and a translation between Japanese and English in order to prove the successful functions of the Prolog programming.

First of all, I use the query, `verb(M,T,G,F,V,[]).`, to elicit all possible verbs and their applicable forms. Because the program contains 306 verbs, the program can elicit all verbs as the output. In Output 1, I will just show the part of the output with the first and second verbs in the program.

Output 1: Partial Output of the First Verb *"agaru"* and the Second Verb *"ageru"*

M=Main Verb Entry, T=Translation, G=Group, F=Form, V=Verb

```
[1] ?-verb(M,T,G,F,V,[]).

Soln: 1
M = agaru;   T = to_go_up;   G = cs;   F = imperfective / present / negative / informal;
V = [agar, - a, - na, - i]
;
Soln: 2
M = agaru;   T = to_go_up;   G = cs;   F = imperfective / past / negative / informal;
V = [agar, - a, - nakat, - ta]
;
Soln: 3
M = agaru;   T = to_go_up;   G = cs;   F = imperfective / present / negative_continuative /
informal;
V = [agar, - a, - naku, - te]
;
Soln: 4
M = agaru;   T = to_go_up;   G = cs;   F = imperfective / present / negative_conditional /
informal
V = [agar, - a, - nake, - reba]
```

;
Soln: 5
M = agaru; T = to_go_up; G = cs; F = imperfective / past / negative_conditional / informal;
V = [agar, - a, - nakat, - ta, - ra]
;
Soln: 6
M = agaru; T = to_go_up; G = cs; F = causative2 / present / affirmative / informal;
V = [agar, - a, - se, - ru]
;
Soln: 7
M = agaru; T = to_go_up; G = cs; F = passive2 / present / affirmative / informal;
V = [agar, - a, - re, - ru]
;
Soln: 8
M = agaru; T = to_go_up; G = cs; F = polite / present / affirmative / formal;
V = [agar, - i, - mas, - u]
;
Soln: 9
M = agaru; T = to_go_up; G = cs; F = polite / past / affirmative / formal;
V = [agar, - i, - mashi, - ta]
;
Soln: 10
M = agaru; T = to_go_up; G = cs; F = polite / present / negative / formal;
V = [agar, - i, - mase, - n]
;
Soln: 11
M = agaru; T = to_go_up; G = cs; F = polite / past / negative / formal;
V = [agar, - i, - mase, - n, - deshita]
;
Soln: 12
M = agaru; T = to_go_up; G = cs; F = polite / past / affirmative_conditional / formal;
V = [agar, - i, - mashi, - ta, - ra]
;
Soln: 13
M = agaru; T = to_go_up; G = cs; F = polite / past / negative_conditional / formal;
V = [agar, - i, - mase, - n, - deshita, - ra]
;

Soln: 14
M = agaru; T = to_go_up; G = cs; F = polite / present / affirmative_volitional / formal;
V = [agar, - i, - masho, - o]
;
Soln: 15
M = agaru; T = to_go_up; G = cs; F = predicative1 / present / affirmative / informal;
V = [agar, - u]
;
Soln: 16
M = agaru; T = to_go_up; G = cs; F = conditional1 / present / affirmative / informal;
V = [agar, - e, - ba]
;
Soln: 17
M = agaru; T = to_go_up; G = cs; F = imperative1 / present / affirmative / informal;
V = [agar, - e]
;
Soln: 18
M = agaru; T = to_go_up; G = cs; F = negative_imperative1 / present / negative /
informal;
V = [agar, - u, - na]
;
Soln: 19
M = agaru; T = to_go_up; G = cs; F = volitional1 / present / affirmative / informal;
V = [agar, - o, - o]
;
Soln: 20
M = agaru; T = to_go_up; G = cs; F = potential1 / present / affirmative / informal;
V = [agar, - e, - ru]
;
Soln: 21
M = agaru; T = to_go_up; G = cs / process1; F = continuative1 / present / affirmative /
informal;
V = [agat, - te]
;
Soln: 22
M = agaru; T = to_go_up; G = cs / process1; F = past_tense1 / past / affirmative /
informal;
V = [agat, - ta]

;
Soln: 23
M = agaru; T = to_go_up; G = cs / process1; F = conditional_past1 / past / affirmative / informal;
V = [agat, - ta, - ra]
;
Soln: 24
M = arasou; T = to_fight; G = cs / irreg1 / wa; F = imperfective / present / negative / informal;
V = [araso, - wa, - na, - i]
….(continued)

The program succeeds in providing all 23 applicable verb forms of the first listed verb *"agaru"* and continues with all the forms of the second listed verb *"arasou."* Thus, the output proves the search function of all verbs in the program is successful.

Next, I will elicit the applicable forms of the specific verbs in each conjugation group: *"kaku"* for a consonant verb, *"taberu"* for a vowel verb, and *"kuru"* and *"suru"* irregular verbs, exemplified in Section 2 with the following queries:

`verb('kaku',T,G,F,V,[]).` : For the verb *"kaku;"*
`verb('taberu',T,G,F,V,[]).` : For the verb *"taberu;"*
`verb('kuru',T,G,F,V,[]).` : For the verb *"kuru;"*
`verb('suru',T,G,F,V,[]).` : For the verb *"suru"*

See appendix 2-5 for the outputs of each verb conjugation.

The program also has a search function to find the feature information of a specific verb form regarding Main verb entry, Translation, Group and Function. Output 2 demonstrates the query to elicit the feature information of the form *"kak-a-na-i."*

Output 2: The Feature Information of the Verb Form of *"kak-a-na-i"*
[1] ?-verb(M,T,G,F,[kak,-a,-na,-i],[]).

Soln: 1
M = kaku; T = to_write; G = cs; F = imperfective / present / negative / informal
;

No More Solutions.

For the convenience for non-native Japanese learners, this program can be used to search the stem and root of each form of the specific verb as well. In order to conduct the verb stem search in the program, the extra rule indicated below is added to the main predicate rules.

```
v_stem(MVE,TRAN,GROUP,FORM)-->v_stem_base(MVE,TRAN,GROUP,FORM/TENSE/POLARI
TY/FORMALITY).
```

Output 3 shows the output of the base verb stems of the verb *"kaku"* in each form.

Output 3: The Base Verb Stem of *"kaku"* in Each Form

```
[1] ?-v_stem('kaku',T,G,F,V_stem,[]).

Soln: 1
T = to_write; G = cs; F = imperfective; V_stem = [kak, - a]
;
Soln: 2
T = to_write; G = cs; F = causative2; V_stem = [kak, - a]
;
Soln: 3
T = to_write; G = cs; F = passive2; V_stem = [kak, - a]
;
Soln: 4
T = to_write; G = cs; F = polite; V_stem = [kak, - i]
;
Soln: 5
T = to_write; G = cs; F = predicative1; V_stem = [kak, - u]
;
Soln: 6
T = to_write; G = cs; F = conditional1; V_stem = [kak, - e]
;
Soln: 7
T = to_write; G = cs; F = imperative1; V_stem = [kak, - e]
;
Soln: 8
T = to_write; G = cs; F = negative_imperative1; V_stem = [kak, - u]
```

;
Soln: 9
T = to_write; G = cs; F = volitional1; V_stem = [kak, - o]
;
Soln: 10
T = to_write; G = cs; F = potential1; V_stem = [kak, - e]
;
Soln: 11
T = to_write; G = cs / process3; F = continuative1; V_stem = [kai]
;
Soln: 12
T = to_write; G = cs / process3; F = past_tense1; V_stem = [kai]
;
Soln: 13
T = to_write; G = cs / process3; F = conditional_past1; V_stem = [kai]
;
No More Solutions.

Output 4 shows the output of the verb root of the verb *"katsu,"* which has four root variations due to *Onbin* Process 1.

Output 4: The Verb Root of *"katsu"*

[1] ?-vroot('katsu',T,G,Vroot,[]).

Soln: 1
T = to_win; G = cs / t; Vroot = [kat]
;
Soln: 2
T = to_win; G = cs / ts; Vroot = [kats]
;
Soln: 3
T = to_win; G = cs / ch; Vroot = [kach]
;
Soln: 4
T = to_win; G = cs / process1; Vroot = [kat]
;
No More Solutions.

The program elicits not only a verb root or a verb stem but also an auxiliary suffix of a specific form. When queried for auxiliary suffixes, the query needs to specify the form with the four various variables with respect to FORM, TENSE, POLARITY, and FORMALITY in order to elicit each type of auxiliary suffix: Derivational suffix, Formality, Polarity, Tense, and Conjunctive suffix. Output 5 shows the sample outputs of the form of auxiliary suffixes.

Output 5: The Formal Past Negative Conditional Form of Auxiliary Suffixes in Order

[Formal past negative conditional form of auxiliary suffix]

[Derivational Suffix]

```
[1] ?-aux_suf_ds(polite/past/negative_conditional/formal,Aux,[]).

Soln: 1
Aux = []
;
No More Solutions.
```

[Formality]

```
[1] ?- aux_suf_formal(polite/past/negative_conditional/formal, Aux,[]).

Soln: 1
Aux = [- mase]
;
No More Solutions.
```

[Polarity]

```
[1] ?- aux_suf_pol(polite/past/negative_conditional/formal,Aux,[]).

Soln: 1
Aux = [- n]
;
No More Solutions.
```

[Tense]

```
[1] ?- aux_suf_tns(polite/past/negative_conditional/formal,Aux,[]).
```

Soln: 1
Aux = [- deshita]
;
No More Solutions.

[Conjunctive]
[1] ?- aux_suf_conj(polite/past/negative_conditional/formal,Aux,[]).

Soln: 1
Aux = [- ra]
;
No More Solutions.

From the result of the query, the auxiliary suffixes of the formal past negative conditional form are *" -mase -n -deshita -ra"* as follows:

- [ ]	- [*mase*]	- [*n*]	- [*deshita*]	- [*ra*]
- **Derivational Suffix**	- **Formality**	- **Polarity**	- **Tense**	- **Conjunction**
- [null]	- [formal]	- [negative]	- [past formal]	- [conditional]

The program also elicits a verb form of a specific verb with specific features. Output 6 shows the output of the verb form of *"oyogu"* in the informal present continuative form.

Output 6: The Verb Form of *"oyogu"* in the Informal Present Continuative Form
[1] ?-verb(oyogu,T,G,continuative2/present/affirmative/informal,V,[]).

Soln: 1
T = to_swim; G = cs / process2 / de / da; V = [oyoi, - de]
;
No More Solutions.

Additionally, the program has a Japanese/English translation function by using a query of seeking a verb root, which is demonstrated in Output 7.

Output 7: Translation of the Verb *"aku"* in English and "wait" in Japanese
[English translation of the verb *"aku"*]

```
[1] ?-vroot(aku,T,G,Vroot,[]).

Soln: 1
T = to_open / intrans; G = cs; Vroot = [ak]
;
Soln: 2
T = to_open / intrans; G = cs / process3; Vroot = [ai]
;
No More Solutions.
```

[Japanese translation of the verb *"wait"*]

```
[1] ?-vroot(M,to_wait,G,Vroot,[]).

Soln: 1
M = matsu; G = cs / t; Vroot = [mat]
;
Soln: 2
M = matsu; G = cs / ts; Vroot = [mats]
;
Soln: 3
M = matsu; G = cs / ch; Vroot = [mach]
;
Soln: 4
M = matsu; G = cs / process1; Vroot = [mat]
;
No More Solutions.
```

The Prolog program of Japanese verb conjugation successfully computes Japanese verb formation and elicits the correct verb formations precisely. Besides the verb forms, this program is useful to search the additional information of a verb formation such as a verb stem, a verb root, and an auxiliary suffix as well as the translation between Japanese and English. The Prolog program of Japanese verb conjugation would provide a great assistance to non-native Japanese learners studying Japanese verb conjugation.

8. CONCLUSIONS AND FUTURE WORK

The two major topics dealt with in this work are a linguistic analysis of Japanese verb conjugation patterns in a morphological verb conjugation model based on the Roman alphabet, and the Prolog programming of Japanese verb conjugation for non-native Japanese learners. In the linguistic analysis, the morphological verb conjugation model was analyzed in terms of three conjugation groups in Japanese verb formation: Consonant verbs, Vowel verbs, and Irregular verbs of *"kuru"* and *"suru."* Except for the irregular verbs, both consonant verbs and vowel verbs are distinguished by the ending phoneme of each verb root. The verb *"kaku"* is categorized to a consonant verb because the root ends with a consonant as *'kak*;*'* and the verb *"taberu"* is categorized to a vowel verb because the root ends with a vowel as *'tabe.'* Because consonant verbs end with a consonant, they require a stem forming suffix (SFS) to facilitate the typical Japanese phoneme sequence rule VCVC in the verb formation.

The classification of Japanese verb conjugation groups seems to be simple, though only consonant verbs make their conjugation pattern complicated due to their typical morphophonemic alternation called an '*Onbin*' process. Within the total 306 Japanese verbs, I found that the *Onbin* process occurs only for consonant verbs; and the *Onbin* process occurs particularly when the followed auxiliary suffix begins with the phoneme '*-t*' in continuative and past tense forms. In this article, I introduced three *Onbin* processes and some minor morphophonemic alternations occurring in a consonant verb conjugation. In Prolog programming, I have classified all verbs into three verb conjugation patterns by assigning a group classification to each verb root and have defined root variations in consonant verbs to compute the *Onbin* processes in order to elicit all verb forms precisely.

With respect to the auxiliary suffixes in Japanese verb formation, I have classified them into four types of auxiliary suffixes: Derivational suffix, Formality, Polarity, Tense, and Conjunctive suffix. All of them are constructed in a certain order to stack up after a verb stem to form a verb. In order to apply the auxiliary suffixes' structure into the word formation tree structure, I have defined the recursive rules to attach these various auxiliary suffixes to a verb stem. The binary hierarchy structure analysis successfully states the relations between a verb stem and auxiliary suffixes in clauses to compute the verb forms precisely in the Prolog programing.

Consequently, the outputs prove the usefulness of the program with the various search functions in Japanese verb conjugation for non-native Japanese learners. For the

further study, I would like to expand this programing to the sentence level in order to compute a Japanese sentence structure in the Prolog program.

REFERENCES:

Bratko, Ivan. (1986). *Prolog programming for artificial intelligence*. Wokingham, England: Addison-Wesley.

Cole Jennifer and Hualde Jose I. (2011). Underlying representations. In Marc van Oostendorp, Colin J. Ewen, Elisabeth Hume & Keren Rice (eds.), *The Blackwell Companion to Phonology.* Oxford: Wiley-Blackwell. 1-26.

Hiro Japanese Center (Compiler). (2013). *600 basic Japanese verbs*: *The essential reference guide*. North Clarendon, VT: Tuttle Publishing.

Iwasaki, Shoichi. (2013). *Japanese*. Amsterdam: Philadelphia, PA : John Benjamins.

Bermúdez-Otero, Ricardo. (2012). Morphology and phonology of exponence. In Jochen Trommer (ed.), *The architecture of grammar and division of labor in exponence.* UK: Oxford University Press. 8-83.

Lange, Roland A. (1998). *501 Japanese verbs: Fully described in all inflections, moods, aspects, and formality levels in a new easy-to-learn format, alphabetically arranged*. Hauppauge, N.Y.: Barron's Educational Series.

Matthews, Clive. (1998). *An introduction to natural language processing through Prolog*. London: Longman.

McCawley, James. (1968). *The phonological component of a grammar of Japanese*. The Hague: Mouton.

Pearson, Bruce L. (1972). Crazy Rules and Natural Rules in Japanese Phonology. *Papers in Japanese Linguistics, v1, n1,* 89-102. Japanese Linguistics Workshop. Berkeley: University of California, Berkeley.

Pereira, C.N. Fernando & Shieber M. Stuart, (1987). *Prolog and natural-language analysis*. Brookline, Massachusetts: Microtome Publishing.

Pinker, Steven. (1994). *The Language Instinct.* New York: Harper.

Sasaki Alam, Y. (2008, January 1). A rule-based morpho-semantic analyzer of the Japanese verb phrases of simple sentences. In Rachel Edita O. Roxas (Ed.) *Proceedings of the 22nd Pacific Asia Conference on Language, Information and Computation.* Paper presented at PACLIC 22 Cebu City, Philippines (101-112). Manila, Philippines: De La Salle University-Manila.

Sobin, Nicholas. (2013). *all_paradigms.pl.* (m.s.)

Sobin, Nicholas. (2013). Computational Linguistics: A Hands-On Introduction. El Paso: The Univeristy of Texas at El Paso.

Volpe, Mark Joseph. (2005). *Japanese Morphology and its Theoretical Consequences: Derivational Morphology in Distributed Morphology*. PhD. Dissertation,.Stony Brook University.

APPENDIX 1: Final Prolog Program of Japanese Verb Conjugation

```
%%% Jap_verb_conj.pl  %%% MVE:Main Verb Entry
%%% sfs:stem forming suffix  %%% aux_suf: auxiliary suffixes
%%% conj:Conjunctive; tns:Tense; pol:Polarity: formal:Formal; ds;Derivational

verb(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_tns(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_conj(FORM/TENSE/POLARITY/FORMALITY).

v_stem_tns(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_pol(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_tns(FORM/TENSE/POLARITY/FORMALITY).

v_stem_pol(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_formal(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_pol(FORM/TENSE/POLARITY/FORMALITY).

v_stem_formal(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_ds(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),
aux_suf_formal(FORM/TENSE/POLARITY/FORMALITY).

v_stem_ds(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
v_stem_base(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY),aux_suf_ds(FORM/T
ENSE/POLARITY/FORMALITY).

v_stem_base(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY) -->
vroot(MVE,TRAN,GROUP), sfs(FORM,GROUP).

v_stem(MVE,TRAN,GROUP,FORM) -->
v_stem_base(MVE,TRAN,GROUP,FORM/TENSE/POLARITY/FORMALITY). %%% Necessary for
Base verb stem search

%%% Verb Roots:
vroot(MVE,TRAN,GROUP) --> [Vroot],{vroot(Vroot,MVE,TRAN,GROUP)}.

%%% Consonant verbs: cs
%%% Consonant verbs -Onbin Process 1
```

```
vroot(agar,'agaru',to_go_up,cs). vroot(agat,'agaru',to_go_up,cs/process1).
vroot(araso,'arasou',to_fight,cs/irreg1/wa).
vroot(arasot,'arasou',to_fight,cs/process1).
vroot(ara,'arau',to_wash,cs/irreg1/wa).
vroot(arat,'arau',to_wash,cs/process1).
vroot(ar,'aru',to_be,cs/ar). vroot([],'aru',to_be,cs/[]).
vroot(at,'aru',to_be,cs/process1).
vroot(atar,'ataru',to_hit/intrans,cs).
vroot(atat,'ataru',to_hit/intrans,cs/process1).
vroot(atsumar,'atsumaru',to_get_together,cs).
vroot(atsumat,'atsumaru',to_get_together,cs/process1).
vroot(a,'au',to_meet,cs/irreg1/wa). vroot(at,'au',to_meet,cs/process1).
vroot(ayamar,'ayamaru',to_apologize,cs).
vroot(ayamat,'ayamaru',to_apologize,cs/process1).
vroot(azukar,'azukaru',to_keep,cs).
vroot(azukat,'azukaru',to_keep,cs/process1).
vroot(butsukar,'butsukaru',to_face,cs).
vroot(butsukat,'butsukaru',to_face,cs/process1).
vroot(chiga,'chigau',to_differ,cs/irreg1/wa).
vroot(chigat,'chigau',to_differ,cs/process1).
vroot(fur,'furu',to_fall/intrans/to_reject/trans,cs).
vroot(fut,'furu',to_fall/intrans/to_reject/trans,cs/process1).
vroot(futor,'futoru',to_gain_weight,cs).
vroot(futot,'futoru',to_gain_weight,cs/process1).
vroot(ganbar,'ganbaru',to_do_best,cs).
vroot(ganbat,'ganbaru',to_do_best,cs/process1).
vroot(gozar,'gozaru',to_be/humble_expression,cs).
vroot(gozat,'gozaru',to_be/humble_expression,cs/process1).
vroot(hair,'hairu',to_enter,cs). vroot(hait,'hairu',to_enter,cs/process1).
vroot(hajimar,'hajimaru',to_begin/intrans,cs).
vroot(hajimat,'hajimaru',to_begin/intrans,cs/process1).
vroot(hakar,'hakaru',to_measure,cs).
vroot(hakat,'hakaru',to_measure,cs/process1).
vroot(hara,'harau',to_pay,cs/irreg1/wa).
vroot(harat,'harau',to_pay,cs/process1).
vroot(har,'haru',to_stick/stretch,cs).
vroot(hat,'haru',to_stick/stretch,cs/process1).
vroot(hashir,'hashiru',to_run,cs).
```

```
vroot(hashit,'hashiru',to_run,cs/process1).
vroot(hayar,'hayaru',to_be_popular,cs).
vroot(hayat,'hayaru',to_be_popular,cs/process1).
vroot(her,'heru',to_decrease/intrans,cs).
vroot(het,'heru',to_decrease/intrans,cs/process1).
vroot(hiro,'hirou',to_pick_up,cs/irreg1/wa).
vroot(hirot,'hirou',to_pick_up,cs/process1).
vroot(ik,'iku',to_go,cs). vroot(it,'iku',to_go,cs/process1).
vroot(inor,'inoru',to_pray,cs). vroot(inot,'inoru',to_pray,cs/process1).
vroot(irasshar,'irassharu',to_go/come/be/honorific,cs/irasshar).
vroot(irassha,'irassharu',to_go/come/be/honorific,cs/irreg2/irassha).
vroot(irasshat,'irassharu',to_go/come/be/honorific,cs/process1).
vroot(ir,'iru',to_need,cs). vroot(it,'iru',to_need,cs/process1).
vroot(i,'iu',to_say,cs/irreg1/wa). vroot(it,'iu',to_say,cs/process1).
vroot(iwa,'iwau',to_celebrate,cs/irreg1/wa).
vroot(iwat,'iwau',to_celebrate,cs/process1).
vroot(kaer,'kaeru',to_return/intrans,cs).
vroot(kaet,'kaeru',to_return/intrans,cs/process1).
vroot(kakar,'kakaru',to_take/hang/intrans,cs).
vroot(kakat,'kakaru',to_take/hang/intrans,cs/process1).
vroot(kama,'kamau',to_mind/care,cs/irreg1/wa).
vroot(kamat,'kamau',to_mind/care,cs/process1).
vroot(kat,'katsu',to_win,cs/t). vroot(kats,'katsu',to_win,cs/ts).
vroot(kach,'katsu',to_win,cs/ch). vroot(kat,'katsu',to_win,cs/process1).
vroot(ka,'kau',to_buy,cs/irreg1/wa). vroot(kat,'kau',to_buy,cs/process1).
vroot(kawar,'kawaru',to_alternate/exchange/replace/intrans,cs).
vroot(kawat,'kawaru',to_alternate/exchange/replace/intrans,cs/process1).
vroot(kimar,'kimaru',to_be_decided/intrans,cs).
vroot(kimat,'kimaru',to_be_decided/intrans,cs/process1).
vroot(kir,'kiru',to_cut/trans,cs).
vroot(kit,'kiru',to_cut/trans,cs/process1).
vroot(komar,'komaru',to_be_in_trouble,cs).
vroot(komat,'komaru',to_be_in_trouble,cs/process1).
vroot(kotowar,'kotowaru',to_refuse,cs).
vroot(kotowat,'kotowaru',to_refuse,cs/process1).
vroot(kumor,'kumoru',to_be_cloudy/gloomy,cs).
vroot(kumot,'kumoru',to_be_cloudy/gloomy,cs/process1).
vroot(machiga,'machigau',to_make_a_mistake/intrans,cs/irreg1/wa).
```

```
vroot(machigat,'machigau',to_make_a_mistake/intrans,cs/process1).
vroot(magar,'magaru',to_bend/turn/intrans,cs).
vroot(magat,'magaru',to_bend/turn/intrans,cs/process1).
vroot(mamor,'mamoru',to_protect,cs).
vroot(mamot,'mamoru',to_protect,cs/process1).
vroot(mania,'maniau',to_be_in_time,cs/irreg1/wa).
vroot(maniat,'maniau',to_be_in_time,cs/process1).
vroot(mat,'matsu',to_wait,cs/t). vroot(mats,'matsu',to_wait,cs/ts).
vroot(mach,'matsu',to_wait,cs/ch). vroot(mat,'matsu',to_wait,cs/process1).
vroot(mawar,'mawaru',to_turn_around/intrans,cs).
vroot(mawat,'mawaru',to_turn_around/intrans,cs/process1).
vroot(mayo,'mayou',to_be_lost/be_in_doubt,cs/irreg1/wa).
vroot(mayot,'mayou',to_be_lost/be_in_doubt,cs/process1).
vroot(medat,'medatsu',to_be_remarkable,cs/t).
vroot(medats,'medatsu',to_be_remarkable,cs/ts).
vroot(medach,'medatsu',to_be_remarkable,cs/ch).
vroot(medat,'medatsu',to_be_remarkable,cs/process1).
vroot(mitsukar,'mitsukaru',to_be_found/intrans,cs).
vroot(mitsukat,'mitsukaru',to_be_found/intrans,cs/process1).
vroot(mookar,'mookaru',to_make_a_profit/intrans,cs).
vroot(mookat,'mookaru',to_make_a_profit/intrans,cs/process1).
vroot(mora,'morau',to_receive/get/trans,cs/irreg1/wa).
vroot(mora,'morau',to_receive/get/trans,cs/process1).
vroot(mot,'motsu',to_have/hold,cs/t).
vroot(mots,'motsu',to_have/hold,cs/ts).
vroot(moch,'motsu',to_have/hold,cs/ch).
vroot(mot,'motsu',to_have/hold,cs/process1).
vroot(nakunar,'nakunaru',to_run_out/die/intrans,cs).
vroot(nakunat,'nakunaru',to_run_out/die/intrans,cs/process1).
vroot(naor,'naoru',to_be_repaired/recover/intrans,cs).
vroot(naot,'naoru',to_be_repaired/recover/intrans,cs/process1).
vroot(nara,'narau',to_learn,cs/irreg1/wa).
vroot(narat,'narau',to_learn,cs/process1).
vroot(nar,'naru',to_become/ring,cs).
vroot(nat,'naru',to_become/ring,cs/process1).
vroot(nasar,'nasaru',to_do/honorific/trans,cs).
vroot(nasat,'nasaru',to_do/honorific/trans,cs/process1).
vroot(nemur,'nemuru',to_sleep,cs).
```

```
vroot(nemut,'nemuru',to_sleep,cs/process1).
vroot(nobor,'noboru',to_climb/rise/intrans,cs).
vroot(nobot,'noboru',to_climb/rise/intrans,cs/process1).
vroot(nokor,'nokoru',to_remain,cs).
vroot(nokot,'nokoru',to_remain,cs/process1).
vroot(nor,'noru',to_ride,cs).      vroot(not,'noru',to_ride,cs/process1).
vroot(nur,'nuru',to_paint,cs). vroot(nut,'nuru',to_paint,cs/process1).
vroot(odor,'odoru',to_dance,cs). vroot(odot,'odoru',to_dance,cs/process1).
vroot(okor,'okoru',to_get_angry/occur/intrans,cs).
vroot(okot,'okoru',to_get_angry/occur/intrans,cs/process1).
vroot(okur,'okuru',to_send/see_off,cs).
vroot(okut,'okuru',to_send/see_off,cs/process1).
vroot(omo,'omou',to_think/trans,cs/irreg1/wa).
vroot(omot,'omou',to_think/trans,cs/process1).
vroot(or,'oru',to_fold/bend/trans,cs).
vroot(ot,'oru',to_fold/bend/trans,cs/process1).
vroot(o,'ou',to_run_after/bear/trans,cs/irreg1/wa).
vroot(ot,'ou',to_run_after/bear/trans,cs/process1).
vroot(owar,'owaru',to_be_finished/intrans,cs).
vroot(owat,'owaru',to_be_finished/intrans,cs/process1).
vroot(sagar,'sagaru',to_fall/drop/intrans,cs).
vroot(sagat,'sagaru',to_fall/drop/intrans,cs/process1).
vroot(saso,'sasou',to_invite,cs/irreg1/wa).
vroot(sasot,'sasou',to_invite,cs/process1).
vroot(shibar,'shibaru',to_tie/bind/trans,cs).
vroot(shibat,'shibaru',to_tie/bind/trans,cs/process1).
vroot(shikar,'shikaru',to_scold,cs).
vroot(shikat,'shikaru',to_scold,cs/process1).
vroot(shimar,'shimaru',to_be_closed/intrans,cs).
vroot(shimat,'shimaru',to_be_closed/intrans,cs/process1).
vroot(shima,'shimau',to_put_back,cs/irreg1/wa).
vroot(shimat,'shimau',to_put_back,cs/process1).
vroot(shir,'shiru',to_know,cs). vroot(shit,'shiru',to_know,cs/process1).
vroot(shitaga,'shitagau',to_follow/obey,cs/irreg1/wa).
vroot(shitagat,'shitagau',to_follow/obey,cs/process1).
vroot(sodat,'sodatsu',to_grow_up,cs/t).
vroot(sodats,'sodatsu',to_grow_up,cs/ts).
vroot(sodach,'sodatsu',to_grow_up,cs/ch).
```

```
vroot(sodat,'sodatsu',to_grow_up,cs/process1).
vroot(su,'suu',to_inhale/suck,cs/irreg1/wa).
vroot(sut,'suu',to_inhale/suck,cs/process1).
vroot(suwar,'suwaru',to_sit_down,cs).
vroot(suwat,'suwaru',to_sit_down,cs/process1).
vroot(tamar,'tamaru',to_be_saved/accumulated/intrans,cs).
vroot(tamat,'tamaru',to_be_saved/accumulated/intrans,cs/process1).
vroot(tasukar,'tasukaru',to_be_saved/rescued/intrans,cs).
vroot(tasukat,'tasukaru',to_be_saved/rescued/intrans,cs/process1).
vroot(tat,'tatsu',to_stand/build/trans,cs/t).
vroot(tats,'tatsu',to_stand/build/trans,cs/ts).
vroot(tach,'tatsu',to_stand/build/trans,cs/ch).
vroot(tat,'tatsu',to_stand/build/trans,cs/process1).
vroot(tayor,'tayoru',to_depend_on,cs).
vroot(tayot,'tayoru',to_depend_on,cs/process1).
vroot(tetsuda,'tetsudau',to_help/assist/trans,cs/irreg1/wa).
vroot(tetsudat,'tetsudau',to_help/assist/trans,cs/process1).
vroot(tomar,'tomaru',to_stop/stay_the_night/fasten/intrans,cs).
vroot(tomat,'tomaru',to_stop/stay_the_night/fasten/intrans,cs/process1).
vroot(tor,'toru',to_catch/steal/take_a_picture/trans,cs).
vroot(tot,'toru',to_catch/steal/take_a_picture/trans,cs/process1).
vroot(toor,'tooru',to_pass_through,cs).
vroot(toot,'tooru',to_pass_through,cs/process1).
vroot(tsuka,'tsukau',to_use/employ,cs/irreg1/wa).
vroot(tsukat,'tsukau',to_use/employ,cs/process1).
vroot(tsukur,'tsukuru',to_make/produce,cs).
vroot(tsukut,'tsukuru',to_make/produce,cs/process1).
vroot(tsutawar,'tsutawaru',to_be_transmitted/introduced/intrans,cs).
vroot(tsutawat,'tsutawaru',to_be_transmitted/introduced/intrans,cs/process1
).
vroot(ur,'uru',to_sell,cs). vroot(ut,'uru',to_sell,cs/process1).
vroot(utaga,'utagau',to_suspect,cs/irreg1/wa).
vroot(utagat,'utagau',to_suspect,cs/process1).
vroot(uta,'utau',to_sing,cs/irreg1/wa).
vroot(utat,'utau',to_sing,cs/process1).
vroot(ut,'utsu',to_hit/shoot,cs/t). vroot(uts,'utsu',to_hit/shoot,cs/ts).
vroot(uch,'utsu',to_hit/shoot,cs/ch).
vroot(ut,'utsu',to_hit/shoot,cs/process1).
```

```
vroot(utsur,'utsuru',to_transfer/be_reflected/photograph/intrans,cs).
vroot(utsut,'utsuru',to_transfer/be_reflected/photogragh/intrans,cs/process
1).
vroot(wakar,'wakaru',to_understand,cs).
vroot(wakat,'wakaru',to_understand,cs/process1).
vroot(wara,'warau',to_laugh,cs/irreg1/wa).
vroot(warat,'warau',to_laugh,cs/process1).
vroot(war,'waru',to_break/trans,cs).
vroot(wat,'waru',to_break/trans,cs/process1).
vroot(watar,'wataru',to_cross/range/intrans,cs).
vroot(watat,'wataru',to_cross/range/intrans,cs/process1).
vroot(yabur,'yaburu',to_tear/defeat/trans,cs).
vroot(yabut,'yaburu',to_tear/defeat/trans,cs/process1).
vroot(yar,'yaru',to_do/give,cs). vroot(yat,'yaru',to_do/give,cs/process1).
vroot(yato,'yatou',to_hire,cs/irreg1/wa).
vroot(yatot,'yatou',to_hire,cs/process1).
vroot(yor,'yoru',to_drop_in/depend_on,cs).
vroot(yot,'yoru',to_drop_in/depend_on,cs/process1).
vroot(yo,'you',to_get_drunk,cs/irreg1/wa).
vroot(yot,'you',to_get_drunk,cs/process1).

%%% Consonant verbs -Onbin Process 2
vroot(asob,'asobu',to_play,cs).
vroot(ason,'asobu',to_play,cs/process2/de/da).
vroot(erab,'erabu',to_choose,cs).
vroot(eran,'erabu',to_choose,cs/process2/de/da).
vroot(fukum,'fukumu',to_contain,cs).
vroot(fukun,'fukumu',to_contain,cs/process2/de/da).
vroot(hakob,'hakobu',to_carry,cs).
vroot(hakon,'hakobu',to_carry,cs/process2/de/da).
vroot(kom,'komu',to_be_crowded,cs).
vroot(kon,'komu',to_be_crowded,cs/process2/de/da).
vroot(musub,'musubu',to_tie/contract,cs).
vroot(musun,'musubu',to_tie/contract,cs/process2/de/da).
vroot(narab,'narabu',to_stand_in_a_line/intrans,cs).
vroot(naran,'narabu',to_stand_in_a_line/intrans,cs/process2/de/da).
vroot(nom,'nomu',to_drink,cs).
vroot(non,'nomu',to_drink,cs/process2/de/da).
```

```
vroot(nusum,'nusumu',to_steal,cs).
vroot(nusun,'nusumu',to_steal,cs/process2/de/da).
vroot(shin,'shinu',to_die/intrans,cs).
vroot(shin,'shinu',to_die/intrans,cs/process2/de/da).
vroot(shizum,'shizumu',to_sink/intrans,cs).
vroot(shizun,'shizumu',to_sink/intrans,cs/process2/de/da).
vroot(sum,'sumu',to_live/end/become_clear,cs).
vroot(sun,'sumu',to_live/end/become_clear,cs/process2/de/da).
vroot(susum,'susumu',to_progress/go_forward/intrans,cs).
vroot(susun,'susumu',to_progress/go_forward/intrans,cs/process2/de/da).
vroot(tanom,'tanomu',to_ask_a_person_to_do,cs).
vroot(tanon,'tanomu',to_ask_a_person_to_do,cs/process2/de/da).
vroot(tanoshim,'tanoshimu',to_enjoy_oneself,cs).
vroot(tanoshin,'tanoshimu',to_enjoy_oneself,cs/process2/de/da).
vroot(tob,'tobu',to_fly/jump/intrans,cs).
vroot(ton,'tobu',to_fly/jump/intrans,cs/process2/de/da).
vroot(tsukam,'tsukamu',to_catch/grasp,cs).
vroot(tsukan,'tsukamu',to_catch/grasp,cs/process2/de/da).
vroot(tsutsum,'tsutsumu',to_wrap,cs).
vroot(tsutsun,'tsutsumu',to_wrap,cs/process2/de/da).
vroot(ukab,'ukabu',to_float,cs).
vroot(ukan,'ukabu',to_float,cs/process2/de/da).
vroot(um,'umu',to_give_birth/yield/produce/trans,cs).
vroot(un,'umu',to_give_birth/yield/produce/trans,cs/process2/de/da).
vroot(yasum,'yasumu',to_rest/sleep/be_absent_from,cs).
vroot(yasun,'yasumu',to_rest/sleep/be_absent_from,cs/process2/de/da).
vroot(yob,'yobu',to_call/invite,cs).
vroot(yon,'yobu',to_call/invite,cs/process2/de/da).
vroot(yom,'yomu',to_read,cs). vroot(yon,'yomu',to_read,cs/process2/de/da).
vroot(yorokob,'yorokobu',to_be_glad/delighted,cs).
vroot(yorokon,'yorokobu',to_be_glad/delighted,cs/process2/de/da).

%%% Consonant verbs -Onbin Process 3
vroot(ak,'aku',to_open/intrans,cs).
vroot(ai,'aku',to_open/intrans,cs/process3).
vroot(arawas,'arawasu',to_appear/trans,cs/s).
vroot(arawash,'arawasu',to_appear/trans,cs/irreg3/sh).
vroot(aruk,'aruku',to_walk,cs). vroot(arui,'aruku',to_walk,cs/process3).
```

```
vroot(chikazuk,'chikazuku',to_approach/intrans,cs).
vroot(chikazui,'chikazuku',to_approach/intrans,cs/process3).
vroot(dak,'daku',to_hug,cs). vroot(dai,'daku',to_hug,cs/process3).
vroot(damas,'damasu',to_deceive,cs/s).
vroot(damash,'damasu',to_deceive,cs/irreg3/sh).
vroot(das,'dasu',to_put_out,cs/s).
vroot(dash,'dasu',to_put_out,cs/irreg3/sh).
vroot(fuk,'fuku',to_wipe/blow,cs).
vroot(fui,'fuku',to_wipe/blow,cs/process3).
vroot(fuyas,'fuyasu',to_increase,cs/s).
vroot(fuyash,'fuyasu',to_increase,cs/irreg3/sh).
vroot(hak,'haku',to_put_on/vomit/sweep,cs).
vroot(hai,'haku',to_put_on/vomit/sweep,cs/process3).
vroot(hanas,'hanasu',to_speak/keep_away/let_go,cs/s).
vroot(hanash,'hanasu',to_speak/keep_away/let-go,cs/irreg3/sh).
vroot(hatarak,'hataraku',to_work,cs).
vroot(hatarai,'hataraku',to_work,cs/process3).
vroot(heras,'herasu',to_decrease/trans,cs/s).
vroot(herash,'herasu',to_decrease/trans,cs/irreg3/sh).
vroot(hik,'hiku',to_pull/play_string_instruments,cs).
vroot(hii,'hiku',to_pull/play_string_instruments,cs/process3).
vroot(isog,'isogu',to_hurry,cs).
vroot(isoi,'isogu',to_hurry,cs/process2/de/da).
vroot(itas,'itasu',to_do/humble,cs/s).
vroot(itash,'itasu',to_do/humble,cs/irreg3/sh).
vroot(kaes,'kaesu',to_return/trans,cs/s).
vroot(kaesh,'kaesu',to_return/trans,cs/irreg3/sh).
vroot(kak,'kaku',to_write,cs). vroot(kai,'kaku',to_write,cs/process3).
vroot(kakus,'kakusu',to_hide,cs/s).
vroot(kakush,'kakusu',to_hide,cs/irreg3/sh).
vroot(kas,'kasu',to_lend,cs/s). vroot(kash,'kasu',to_lend,cs/irreg3/sh).
vroot(kawakas,'kawakasu',to_dry/trans,cs/s).
vroot(kawakash,'kawakasu',to_dry/trans,cs/irreg3/sh).
vroot(kawak,'kawaku',to_get_dry/intrans,cs).
vroot(kawai,'kawaku',to_get_dry/intrans,cs/process3).
vroot(kes,'kesu',to_erase/disappear/trans,cs/s).
vroot(kesh,'kesu',to_erase/disappear/trans,cs/irreg3/sh).
vroot(kik,'kiku',to_listen/trans,cs).
```

```
vroot(kii,'kiku',to_listen/trans,cs/process3).
vroot(koros,'korosu',to_kill,cs/s).
vroot(korosh,'korosu',to_kill,cs/irreg3/sh).
vroot(kos,'kosu',to_cross,cs/s). vroot(kosh,'kosu',to_cross,cs/irreg3/sh).
vroot(kowas,'kowasu',to_break/trans,cs/s).
vroot(kowash,'kowasu',to_break/trans,cs/irreg3/sh).
vroot(mak,'maku',to_bind/seed,cs).
vroot(mai,'maku',to_bind/seed,cs/process3).
vroot(mawas,'mawasu',to_turn/trans,cs/s).
vroot(mawash,'mawasu',to_turn/trans,cs/irreg3/sh).
vroot(migak,'migaku',to_polish,cs).
vroot(migai,'migaku',to_polish,cs/process3).
vroot(moras,'morasu',to_let_leak_out/trans,cs/s).
vroot(morash,'morasu',to_let_leak_out/trans,cs/irreg3/sh).
vroot(muk,'muku',to_look_toward/peel,cs).
vroot(mui,'muku',to_look_toward/peel,cs/process3).
vroot(nak,'naku',to_cry/bark,cs).
vroot(nai,'naku',to_cry/bark,cs/process3).
vroot(nakus,'nakusu',to_lose/trans,cs/s).
vroot(nakush,'nakusu',to_lose/trans,cs/irreg3/sh).
vroot(naos,'naosu',to_fix/cure/trans,cs/s).
vroot(naosh,'naosu',to_fix/cure/trans,cs/irreg3/sh).
vroot(nobas,'nobasu',to_stretch/postpone/trans,cs/s).
vroot(nobash,'nobasu',to_stretch/postpone/trans,cs/irreg3/sh).
vroot(nokos,'nokosu',to_save/leave,cs/s).
vroot(nokosh,'nokosu',to_save/leave,cs/irreg3/sh).
vroot(nozok,'nozoku',to_omit/peep_in,cs).
vroot(nozoi,'nozoku',to_omit/peep_in,cs/process3).
vroot(nug,'nugu',to_take_off_clothes,cs).
vroot(nui,'nugu',to_take_off_clothes,cs/process2/de/da).
vroot(odorok,'odoroku',to_be_surprised/intrans,cs).
vroot(odoroi,'odoroku',to_be_surprised,cs/process3).
vroot(okos,'okosu',to_wake_up/happen/trans,cs/s).
vroot(okosh,'okosu',to_wake_up/happen/trans,cs/irreg3/sh).
vroot(ok,'oku',to_put/leave/trans,cs).
vroot(oi,'oku',to_put/leave/trans,cs/process3).
vroot(os,'osu',to_push,cs/s). vroot(osh,'osu',to_push,cs/irreg3/sh).
vroot(otos,'otosu',to_drop/remove,cs/s).
```

```
vroot(otosh,'otosu',to_drop/remove,cs/irreg3/sh).
vroot(oyog,'oyogu',to_swim,cs).
vroot(oyoi,'oyogu',to_swim,cs/process2/de/da).
vroot(sagas,'sagasu',to_look_for,cs/s).
vroot(sagash,'sagasu',to_look_for,cs/irreg3/sh).
vroot(sak,'saku',to_bloom/tear/trans,cs).
vroot(sai,'saku',to_bloom/tear/trans,cs/process3).
vroot(sas,'sasu',to_sting/point_to/insert,cs/s).
vroot(sash,'sasu',to_sting/point_to/insert,cs/irreg3/sh).
vroot(sawag,'sawagu',to_be_noisy,cs).
vroot(sawai,'sawagu',to_be_noisy,cs/process2/de/da).
vroot(shik,'shiku',to_spread/lay/promulgate,cs).
vroot(shii,'shiku',to_spread/lay/promulgate,cs/process3).
vroot(shimes,'shimesu',to_show,cs/s).
vroot(shimesh,'shimesu',to_show,cs/irreg3/sh).
vroot(sugos,'sugosu',to_spend_time,cs/s).
vroot(sugosh,'sugosu',to_spend_time,cs/irreg3/sh).
vroot(suk,'suku',to_like/trans/be_not_crowded/intrans,cs).
vroot(sui,'suku',to_like/trans/be_not_crowded/intrans,cs/process3).
vroot(tames,'tamesu',to_test/try,cs/s).
vroot(tamesh,'tamesu',to_test/try,cs/irreg3/sh).
vroot(taos,'taosu',to_defeat/knock_down,cs/s).
vroot(taosh,'taosu',to_defeat/knock_down,cs/irreg3/sh).
vroot(tobas,'tobasu',to_let_fly/skip/trans,cs/s).
vroot(tobash,'tobasu',to_let_fly/skip/trans,cs/irreg3/sh).
vroot(todok,'todoku',to_reach/arrive/intrans,cs).
vroot(todoi,'todoku',to_reach/arrive/intrans,cs/process3).
vroot(tok,'toku',to_untie/solve/dissolve/trans,cs).
vroot(toi,'toku',to_untie/solve/dissolve,cs/process3).
vroot(tsuk,'tsuku',to_arrive/stick_to/be_lighted/intrans,cs).
vroot(tsui,'tsuku',to_arrive/stick_to/be_lighted/intrans,cs/process3).
vroot(tsuzuk,'tsuzuku',to_continue/intrans,cs).
vroot(tsuzui,'tsuzuku',to_continue/intrans,cs/process3).
vroot(ugokas,'ugokasu',to_move/operate/trans,cs/s).
vroot(ugokash,'ugokasu',to_move/operate/trans,cs/irreg3/sh).
vroot(ugok,'ugoku',to_move/intrans,cs).
vroot(ugoi,'ugoku',to_move/intrans,cs/process3).
vroot(utsus,'utsusu',to_transfer/copy/reflect/trans,cs/s).
```

```
vroot(utsush,'utsusu',to_transfer/copy/reflect/trans,cs/irreg3/sh).
vroot(watas,'watasu',to_hand,cs/s).
vroot(watash,'watasu',to_hand,cs/irreg3/sh).
vroot(yak,'yaku',to_grill/envy/trans,cs).
vroot(yai,'yaku',to_grill/envy/trans,cs/process3).
vroot(yurus,'yurusu',to_forgive/permit,cs/s).
vroot(yurush,'yurusu',to_forgive/permit,cs/irreg3/sh).

%%% Vowel verbs: vw
vroot(age,'ageru',to_raise,vw).
vroot(ake,'akeru',to_open/trans,vw).
vroot(akirame,'akirameru',to_give_up,vw).
vroot(aki,'akiru',to_be_tired_of,vw).
vroot(araware,'arawareru',to_appear/intrans,vw).
vroot(ate,'ateru',to_hit/trans,vw).
vroot(atsume,'atsumeru',to_collect,vw).
vroot(azuke,'azukeru',to_deposit,vw).
vroot(butsuke,'butsukeru',to_hit_against/trans,vw).
vroot(chikazuke,'chikazukeru',to_allow_to_approach/trans,vw).
vroot(dekake,'dekakeru',to_go_out,vw).
vroot(deki,'dekiru',to_be_able_to,vw).
vroot(de,'deru',to_come_out,vw).
vroot(fue,'fueru',to_increase,vw).
vroot(fukume,'fukumeru',to_include/trans,vw).
vroot(fure,'fureru',to_touch,vw).
vroot(hajime,'hajimeru',to_begin/trans,vw).
vroot(hare,'hareru',to_clear_up/swell,vw).
vroot(home,'homeru',to_praise,vw).
vroot(iki,'ikiru',to_live,vw).
vroot(ire,'ireru',to_put_in/trans,vw).
vroot(i,'iru',to_be/stay,vw).
vroot(kae,'kaeru',to_change,vw).
vroot(kake,'kakeru',to_hang/call/lock/trans,vw).
vroot(kakure,'kakureru',to_hide,vw).
vroot(kangae,'kangaeru',to_think,vw).
vroot(kanji,'kanjiru',to_feel,vw).
vroot(kari,'kariru',to_borrow,vw).
vroot(katazuke,'katazukeru',to_tidy_up/finish,vw).
```

```
vroot(kazoe,'kazoeru',to_count,vw).
vroot(kie,'kieru',to_disappear/intrans,vw).
vroot(kikoe,'kikoeru',to_hear/intrans,vw).
vroot(kime,'kimeru',to_decide/trans,vw).
vroot(kire,'kireru',to_cut/intrans,vw).
vroot(ki,'kiru',to_wear,vw).
vroot(koe,'koeru',to_go_over,vw).
vroot(kotae,'kotaeru',to_respond,vw).
vroot(koware,'kowareru',to_be_broken/intrans,vw).
vroot(kurabe,'kuraberu',to_compare,vw).
vroot(kure,'kureru',to_give/get_dark,vw).
vroot(kuwae,'kuwaeru',to_add/hold_in_mouth,vw).
vroot(machigae,'machigaeru',to_make_a_mistake/trans,vw).
vroot(mage,'mageru',to_bend/twist/trans,vw).
vroot(makase,'makaseru',to_entrust_to,vw).
vroot(make,'makeru',to_lose,vw).
vroot(mie,'mieru',to_be_seen/intrans,vw).
vroot(mi,'miru',to_look/trans,vw).
vroot(mise,'miseru',to_show,vw).
vroot(mitome,'mitomeru',to_permit/admit,vw).
vroot(mitsuke,'mitsukeru',to_find/trans,vw).
vroot(mooke,'mookeru',to_make_a_profit/trans,vw).
vroot(more,'moreru',to_leak_out/intrans,vw).
vroot(motome,'motomeru',to_demand,vw).
vroot(muke,'mukeru',to_turn_toward,vw).
vroot(nage,'nageru',to_throw/pitch,vw).
vroot(narabe,'naraberu',to_line_up/arrange/trans,vw).
vroot(nare,'nareru',to_get_used_to/intrans,vw).
vroot(ne,'neru',to_go_to_bed,vw).
vroot(nige,'nigeru',to_escape,vw).
vroot(ni,'niru',to_boil/resemble,vw).
vroot(nobi,'nobiru',to_grow/postpone/intrans,vw).
vroot(nure,'nureru',to_get_wet/intrans,vw).
vroot(oboe,'oboeru',to_remember/memorize,vw).
vroot(ochi,'ochiru',to_fall,vw).
vroot(oki,'okiru',to_wake_up,vw).
vroot(okure,'okureru',to_be_delayed/be_far_behind/intrans,vw).
vroot(ore,'oreru',to_be_broken/be_folded/intrans,vw).
```

```
vroot(ori,'oriru',to_get_off/come_down/intrans,vw).
vroot(osae,'osaeru',to_catch/suppress/reserve,vw).
vroot(oshie,'oshieru',to_teach,vw).
vroot(sage,'sageru',to_lower/pull_back/trans,vw).
vroot(sake,'sakeru',to_split/intrans/to_avoid/trans,vw).
vroot(sasae,'sasaeru',to_support,vw).
vroot(shime,'shimeru',to_close/trans,vw).
vroot(shinji,'shinjiru',to_believe,vw).
vroot(shirabe,'shiraberu',to_investigate,vw).
vroot(shirase,'shiraseru',to_inform,vw).
vroot(sodate,'sodateru',to_raise/trans,vw).
vroot(sugi,'sugiru',to_pass/intrans,vw).
vroot(susume,'susumeru',to_advance/recommend/trans,vw).
vroot(sute,'suteru',to_throw_away,vw).
vroot(tabe,'taberu',to_eat,vw).
vroot(tame,'tameru',to_save/accumulate/trans,vw).
vroot(taore,'taoreru',to_fall_down/intrans,vw).
vroot(tari,'tariru',to_be_enough/be_sufficient,vw).
vroot(tashikame,'tashikameru',to_check/verify,vw).
vroot(tasuke,'tasukeru',to_help/rescue/trans,vw).
vroot(tate,'tateru',to_stand/establish/build/trans,vw).
vroot(tazune,'tazuneru',to_ask/visit,vw).
vroot(todoke,'todokeru',to_deliver/notify/trans,vw).
vroot(toke,'tokeru',to_be_solved/melt/intrans,vw).
vroot(tome,'tomeru',to_stop/give_lodging/fasten/trans,wv).
vroot(tore,'toreru',to_come_off/be_taken/intrans,vw).
vroot(tsuuji,'tsuujiru',to_connect_with/be_well_informed,vw).
vroot(tsukamae,'tsukamaeru',to_catch/arrest,vw).
vroot(tsukare,'tsukareru',to_be_tired/intrans,vw).
vroot(tsuke,'tsukeru',to_put_on/trans,vw).
vroot(tsutae,'tsutaeru',to_convey/transmit/trans,vw).
vroot(tsuzuke,'tsuzukeru',to_continue/trans,vw).
vroot(umare,'umareru',to_be_born/intrans,vw).
vroot(wakare,'wakareru',to_separate_from/intrans,vw).
vroot(wake,'wakeru',to_divide/trans,vw).
vroot(ware,'wareru',to_break/intrans,vw).
vroot(wasure,'wasureru',to_forget,vw).
vroot(yabure,'yabureru',to_tear/be_defeated/intrans,vw).
```

```
vroot(yake,'yakeru',to_be_burned/jealous/intrans,vw).
vroot(yame,'yameru',to_abandon/resign,vw).
vroot(yase,'yaseru',to_lose_weight,vw).

%%% "kuru" irregular verb: irreg
vroot(ko,'kuru',to_come,irreg/ko). vroot(ki,'kuru',to_come,irreg/ki).
vroot(ku,'kuru',to_come,irreg/ku).

%%% "suru" irregular verb: irreg
vroot(sa,'suru',to_do,irreg/sa).    vroot(shi,'suru',to_do,irreg/sh).
vroot(su,'suru',to_do,irreg/su).     vroot(de,'suru',to_do,irreg/de).

%%% stem forming suffixs: sfs

sfs(imperfective,cs) --> [-a].
sfs(imperfective,cs/irreg1/wa) --> [-wa].
sfs(imperfective,cs/s) --> [-a].
sfs(imperfective,cs/t) --> [-a].
sfs(imperfective,cs/irasshar) --> [-a].
sfs(imperfective,cs/[]) --> [].
sfs(imperfective,vw) --> [].
sfs(imperfective,irreg/ko) --> [].
sfs(imperfective,irreg/sh) --> [].

sfs(causative2,cs) --> [-a].
sfs(causative2,cs/irreg1/wa) --> [-wa].
sfs(causative2,cs/s) --> [-a].
sfs(causative2,cs/t) --> [-a].
sfs(causative1,vw) --> [].
sfs(causative1,irreg/ko) --> [].
sfs(causative2,irreg/sa) --> [].

sfs(passive2,cs) --> [-a].
sfs(passive2,cs/irreg1/wa) --> [-wa].
sfs(passive2,cs/s) --> [-a].
sfs(passive2,cs/t) --> [-a].
sfs(passive2,cs/irasshar) --> [-a].
sfs(passive1,vw) --> [].
```

```
sfs(passive1,irreg/ko) --> [].
sfs(passive2,irreg/sa) --> [].

sfs(polite,cs) --> [-i].
sfs(polite,cs/irreg1/wa) --> [-i].
sfs(polite,cs/irreg3/sh) --> [-i].
sfs(polite,cs/ch) --> [-i].
sfs(polite,cs/ar) --> [-i].
sfs(polite,cs/irreg2/irassha) --> [-i].
sfs(polite,vw) --> [].
sfs(polite,irreg/ki) --> [].
sfs(polite,irreg/sh) --> [].

sfs(continuative1,cs/process1) --> [].
sfs(continuative1,cs/process3) --> [].
sfs(continuative2,cs/process2/de/da) --> [].
sfs(continuative1,cs/irreg3/sh) --> [-i].
sfs(continuative1,vw) --> [].
sfs(continuative1,irreg/ki) --> [].
sfs(continuative1,irreg/sh) --> [].

sfs(past_tense1,cs/process1) --> [].
sfs(past_tense1,cs/process3) --> [].
sfs(past_tense2,cs/process2/de/da) --> [].
sfs(past_tense1,cs/irreg3/sh) --> [-i].
sfs(past_tense1,vw) --> [].
sfs(past_tense1,irreg/ki) --> [].
sfs(past_tense1,irreg/sh) --> [].

sfs(conditional_past1,cs/process1) --> [].
sfs(conditional_past1,cs/process3) --> [].
sfs(conditional_past2,cs/process2/de/da) --> [].
sfs(conditional_past1,cs/irreg3/sh) --> [-i].
sfs(conditional_past1,vw) --> [].
sfs(conditional_past1,irreg/ki) --> [].
sfs(conditional_past1,irreg/sh) --> [].

sfs(predicative1,cs) --> [-u].
```

```
sfs(predicative1,cs/irreg1/wa) --> [-u].
sfs(predicative1,cs/s) --> [-u].
sfs(predicative1,cs/ts) --> [-u].
sfs(predicative1,cs/ar) --> [-u].
sfs(predicative1,cs/irasshar) --> [-u].
sfs(predicative2,vw) --> [].
sfs(predicative2,irreg/ku) --> [].
sfs(predicative2,irreg/su) --> [].

sfs(conditional1,cs) --> [-e].
sfs(conditional1,cs/irreg1/wa) --> [-e].
sfs(conditional1,cs/s) --> [-e].
sfs(conditional1,cs/t) --> [-e].
sfs(conditional1,cs/ar) --> [-e].
sfs(conditional1,cs/irasshar) --> [-e].
sfs(conditional2,vw) --> [].
sfs(conditional2,irreg/ku) --> [].
sfs(conditional2,irreg/su) --> [].

sfs(imperative1,cs) --> [-e].
sfs(imperative1,cs/irreg1/wa) --> [-e].
sfs(imperative1,cs/s) --> [-e].
sfs(imperative1,cs/t) --> [-e].
sfs(imperative2,vw) --> [].
sfs(imperative3,irreg/ko) --> [].
sfs(imperative2,irreg/sh) --> [].

sfs(negative_imperative1,cs) --> [-u].
sfs(negative_imperative1,cs/irreg1/wa) --> [-u].
sfs(negative_imperative1,cs/s) --> [-u].
sfs(negative_imperative1,cs/ts) --> [-u].
sfs(negative_imperative2,vw) --> [].
sfs(negative_imperative2,irreg/ku) --> [].
sfs(negative_imperative2,irreg/su) --> [].

sfs(volitional1,cs) --> [-o].
sfs(volitional1,cs/irreg1/wa) --> [-o].
sfs(volitional1,cs/s) --> [-o].
```

```
sfs(volitional1,cs/t) --> [-o].
sfs(volitional1,cs/irasshar) --> [-o].
sfs(volitional2,vw) --> [].
sfs(volitional2,irreg/ko) --> [].
sfs(volitional2,irreg/sh) --> [].

sfs(potential1,cs) --> [-e].
sfs(potential1,cs/irreg1/wa) --> [-e].
sfs(potential1,cs/s) --> [-e].
sfs(potential1,cs/t) --> [-e].
sfs(potential1,cs/irasshar) --> [-e].
sfs(potential2,vw) --> [].
sfs(potential3,vw) --> [].
sfs(potential2,irreg/ko) --> [].
sfs(potential4,irreg/de) --> [].

%%% aux_suffix
%%% Derivational Suffix (ds)
aux_suf_ds(causative1/present/affirmative/informal)-->[-sase].%causative
form
aux_suf_ds(causative2/present/affirmative/informal)-->[-se].

aux_suf_ds(passive1/present/affirmative/informal)-->[-rare].%passive form
aux_suf_ds(passive2/present/affirmative/informal)-->[-re].

aux_suf_ds(potential1/present/affirmative/informal)-->[].%potential form
aux_suf_ds(potential2/present/affirmative/informal)-->[-rare].
aux_suf_ds(potential3/present/affirmative/informal)-->[-re].
aux_suf_ds(potential4/present/affirmative/informal)-->[-ki].

aux_suf_ds(polite/present/affirmative/formal)-->[].%formal form
aux_suf_ds(polite/past/affirmative/formal)-->[].
aux_suf_ds(polite/present/negative/formal)-->[].
aux_suf_ds(polite/past/negative/formal)-->[].
aux_suf_ds(polite/past/affirmative_conditional/formal)-->[].
aux_suf_ds(polite/past/negative_conditional/formal)-->[].
aux_suf_ds(polite/present/affirmative_volitional/formal)-->[].
```

```
aux_suf_ds(imperfective/present/negative/informal)-->[].%imperfective form
aux_suf_ds(imperfective/past/negative/informal)-->[].
aux_suf_ds(imperfective/present/negative_continuative/informal)-->[].
aux_suf_ds(imperfective/present/negative_conditional/informal)-->[].
aux_suf_ds(imperfective/past/negative_conditional/informal)-->[].

aux_suf_ds(past_tense1/past/affirmative/informal)-->[].%past-tense form
aux_suf_ds(past_tense2/past/affirmative/informal)-->[].

aux_suf_ds(conditional_past1/past/affirmative/informal)-->[].%conditional_p
ast form
aux_suf_ds(conditional_past2/past/affirmative/informal)-->[].

aux_suf_ds(predicative1/present/affirmative/informal)-->[].%predicative
form
aux_suf_ds(predicative2/present/affirmative/informal)-->[].

aux_suf_ds(continuative1/present/affirmative/informal)-->[].%continuative
form
aux_suf_ds(continuative2/present/affirmative/informal)-->[].

aux_suf_ds(conditional2/present/affirmative/informal)-->[].%conditional
form
aux_suf_ds(conditional1/present/affirmative/informal)-->[].

aux_suf_ds(imperative1/present/affirmative/informal)-->[].%imperative form
aux_suf_ds(imperative2/present/affirmative/informal)-->[].
aux_suf_ds(imperative3/present/affirmative/informal)-->[].

aux_suf_ds(negative_imperative1/present/negative/informal)-->[].%imperative
/negative form
aux_suf_ds(negative_imperative2/present/negative/informal)-->[].

aux_suf_ds(volitional1/present/affirmative/informal)-->[].%volitional form
aux_suf_ds(volitional2/present/affirmative/informal)-->[].

%%% Formality (formal)
aux_suf_formal(causative1/present/affirmative/informal)-->[].%causative
```

```
form
aux_suf_formal(causative2/present/affirmative/informal)-->[].

aux_suf_formal(passive1/present/affirmative/informal)-->[].%passive form
aux_suf_formal(passive2/present/affirmative/informal)-->[].

aux_suf_formal(potential1/present/affirmative/informal)-->[].%potential
form
aux_suf_formal(potential2/present/affirmative/informal)-->[].
aux_suf_formal(potential3/present/affirmative/informal)-->[].
aux_suf_formal(potential4/present/affirmative/informal)-->[].

aux_suf_formal(polite/present/affirmative/formal)-->[-mas].%formal form
aux_suf_formal(polite/past/affirmative/formal)-->[-mashi].
aux_suf_formal(polite/present/negative/formal)-->[-mase].
aux_suf_formal(polite/past/negative/formal)-->[-mase].
aux_suf_formal(polite/past/affirmative_conditional/formal)-->[-mashi].
aux_suf_formal(polite/past/negative_conditional/formal)-->[-mase].
aux_suf_formal(polite/present/affirmative_volitional/formal)-->[-masho].

aux_suf_formal(imperfective/present/negative/informal)-->[].%imperfective
form
aux_suf_formal(imperfective/past/negative/informal)-->[].
aux_suf_formal(imperfective/present/negative_continuative/informal)-->[].
aux_suf_formal(imperfective/present/negative_conditional/informal)-->[].
aux_suf_formal(imperfective/past/negative_conditional/informal)-->[].

aux_suf_formal(past_tense1/past/affirmative/informal)-->[].%past-tense form
aux_suf_formal(past_tense2/past/affirmative/informal)-->[].

aux_suf_formal(conditional_past1/past/affirmative/informal)-->[].%condition
al_past form
aux_suf_formal(conditional_past2/past/affirmative/informal)-->[].

aux_suf_formal(predicative1/present/affirmative/informal)-->[].%predicative
form
aux_suf_formal(predicative2/present/affirmative/informal)-->[].
```

```
aux_suf_formal(continuative1/present/affirmative/informal)-->[].%continuati
ve form
aux_suf_formal(continuative2/present/affirmative/informal)-->[].

aux_suf_formal(conditional2/present/affirmative/informal)-->[].%conditional
form
aux_suf_formal(conditional1/present/affirmative/informal)-->[].

aux_suf_formal(imperative1/present/affirmative/informal)-->[].%imperative
form
aux_suf_formal(imperative2/present/affirmative/informal)-->[].
aux_suf_formal(imperative3/present/affirmative/informal)-->[].

aux_suf_formal(negative_imperative1/present/negative/informal)-->[].%impera
tive/negative form
aux_suf_formal(negative_imperative2/present/negative/informal)-->[].

aux_suf_formal(volitional1/present/affirmative/informal)-->[].%volitional
form
aux_suf_formal(volitional2/present/affirmative/informal)-->[].

%%% Polarity (pol)
aux_suf_pol(causative1/present/affirmative/informal)-->[].%causative form
aux_suf_pol(causative2/present/affirmative/informal)-->[].

aux_suf_pol(passive1/present/affirmative/informal)-->[].%passive form
aux_suf_pol(passive2/present/affirmative/informal)-->[].

aux_suf_pol(potential1/present/affirmative/informal)-->[].%potential form
aux_suf_pol(potential2/present/affirmative/informal)-->[].
aux_suf_pol(potential3/present/affirmative/informal)-->[].
aux_suf_pol(potential4/present/affirmative/informal)-->[].

aux_suf_pol(polite/present/affirmative/formal)-->[].%formal form
aux_suf_pol(polite/past/affirmative/formal)-->[].
aux_suf_pol(polite/present/negative/formal)-->[-n].
aux_suf_pol(polite/past/negative/formal)-->[-n].
aux_suf_pol(polite/past/affirmative_conditional/formal)-->[].
```

```
aux_suf_pol(polite/past/negative_conditional/formal)-->[-n].
aux_suf_pol(polite/present/affirmative_volitional/formal)-->[].

aux_suf_pol(imperfective/present/negative/informal)-->[-na].%imperfective
form
aux_suf_pol(imperfective/past/negative/informal)-->[-nakat].
aux_suf_pol(imperfective/present/negative_continuative/informal)-->[-naku].
aux_suf_pol(imperfective/present/negative_conditional/informal)-->[-nake].
aux_suf_pol(imperfective/past/negative_conditional/informal)-->[-nakat].

aux_suf_pol(past_tense1/past/affirmative/informal)-->[].%past-tense form
aux_suf_pol(past_tense2/past/affirmative/informal)-->[].

aux_suf_pol(conditional_past1/past/affirmative/informal)-->[].%conditional_
past form
aux_suf_pol(conditional_past2/past/affirmative/informal)-->[].

aux_suf_pol(predicative1/present/affirmative/informal)-->[].%predicative
form
aux_suf_pol(predicative2/present/affirmative/informal)-->[].

aux_suf_pol(continuative1/present/affirmative/informal)-->[].%continuative
form
aux_suf_pol(continuative2/present/affirmative/informal)-->[].

aux_suf_pol(conditional2/present/affirmative/informal)-->[].%conditional
form
aux_suf_pol(conditional1/present/affirmative/informal)-->[].

aux_suf_pol(imperative1/present/affirmative/informal)-->[].%imperative form
aux_suf_pol(imperative2/present/affirmative/informal)-->[].
aux_suf_pol(imperative3/present/affirmative/informal)-->[].

aux_suf_pol(negative_imperative1/present/negative/informal)-->[-na].%impera
tive/negative form
aux_suf_pol(negative_imperative2/present/negative/informal)-->[].

aux_suf_pol(volitional1/present/affirmative/informal)-->[].%volitional form
```

```
aux_suf_pol(volitional2/present/affirmative/informal)-->[].

%%% Tense (tns)
aux_suf_tns(causative1/present/affirmative/informal)-->[-ru].%causative
form
aux_suf_tns(causative2/present/affirmative/informal)-->[-ru].

aux_suf_tns(passive1/present/affirmative/informal)-->[-ru].%passive form
aux_suf_tns(passive2/present/affirmative/informal)-->[-ru].

aux_suf_tns(potential1/present/affirmative/informal)-->[-ru].%potential
form
aux_suf_tns(potential2/present/affirmative/informal)-->[-ru].
aux_suf_tns(potential3/present/affirmative/informal)-->[-ru].
aux_suf_tns(potential4/present/affirmative/informal)-->[-ru].

aux_suf_tns(polite/present/affirmative/formal)-->[-u].%formal form
aux_suf_tns(polite/past/affirmative/formal)-->[-ta].
aux_suf_tns(polite/present/negative/formal)-->[].
aux_suf_tns(polite/past/negative/formal)-->[-deshita].
aux_suf_tns(polite/past/affirmative_conditional/formal)-->[-ta].
aux_suf_tns(polite/past/negative_conditional/formal)-->[-deshita].
aux_suf_tns(polite/present/affirmative_volitional/formal)-->[].

aux_suf_tns(imperfective/present/negative/informal)-->[-i].%imperfective
form
aux_suf_tns(imperfective/past/negative/informal)-->[-ta].
aux_suf_tns(imperfective/present/negative_continuative/informal)-->[].
aux_suf_tns(imperfective/present/negative_conditional/informal)-->[].
aux_suf_tns(imperfective/past/negative_conditional/informal)-->[-ta].

aux_suf_tns(past_tense1/past/affirmative/informal)-->[-ta].%past-tense form
aux_suf_tns(past_tense2/past/affirmative/informal)-->[-da].

aux_suf_tns(conditional_past1/past/affirmative/informal)-->[-ta].%condition
al_past form
aux_suf_tns(conditional_past2/past/affirmative/informal)-->[-da].
```

```
aux_suf_tns(predicative1/present/affirmative/informal)-->[].%predicative
form
aux_suf_tns(predicative2/present/affirmative/informal)-->[-ru].

aux_suf_tns(continuative1/present/affirmative/informal)-->[].%continuative
form
aux_suf_tns(continuative2/present/affirmative/informal)-->[].

aux_suf_tns(conditional2/present/affirmative/informal)-->[].%conditional
form
aux_suf_tns(conditional1/present/affirmative/informal)-->[].

aux_suf_tns(imperative1/present/affirmative/informal)-->[]. %imperative form
aux_suf_tns(imperative2/present/affirmative/informal)-->[].
aux_suf_tns(imperative3/present/affirmative/informal)-->[].

aux_suf_tns(negative_imperative1/present/negative/informal)-->[].
%imperative/negative form
aux_suf_tns(negative_imperative2/present/negative/informal)-->[-runa].

aux_suf_tns(volitional1/present/affirmative/informal)-->[].%volitional form
aux_suf_tns(volitional2/present/affirmative/informal)-->[].

%%% Conjunctive Suffix (conj)
aux_suf_conj(causative1/present/affirmative/informal)-->[].%causative form
aux_suf_conj(causative2/present/affirmative/informal)-->[].

aux_suf_conj(passive1/present/affirmative/informal)-->[].%passive form
aux_suf_conj(passive2/present/affirmative/informal)-->[].

aux_suf_conj(potential1/present/affirmative/informal)-->[].%potential form
aux_suf_conj(potential2/present/affirmative/informal)-->[].
aux_suf_conj(potential3/present/affirmative/informal)-->[].
aux_suf_conj(potential4/present/affirmative/informal)-->[].

aux_suf_conj(polite/present/affirmative/formal)-->[].%formal form
aux_suf_conj(polite/past/affirmative/formal)-->[].
aux_suf_conj(polite/present/negative/formal)-->[].
```

```
aux_suf_conj(polite/past/negative/formal)-->[].
aux_suf_conj(polite/past/affirmative_conditional/formal)-->[-ra].
aux_suf_conj(polite/past/negative_conditional/formal)-->[-ra].
aux_suf_conj(polite/present/affirmative_volitional/formal)-->[-o].

aux_suf_conj(imperfective/present/negative/informal)-->[].%imperfective
form
aux_suf_conj(imperfective/past/negative/informal)-->[].
aux_suf_conj(imperfective/present/negative_continuative/informal)-->[-te].
aux_suf_conj(imperfective/present/negative_conditional/informal)-->[-reba].
aux_suf_conj(imperfective/past/negative_conditional/informal)-->[-ra].

aux_suf_conj(past_tense1/past/affirmative/informal)-->[].%past-tense form
aux_suf_conj(past_tense2/past/affirmative/informal)-->[].

aux_suf_conj(conditional_past1/past/affirmative/informal)-->[-ra].%conditio
nal_past form
aux_suf_conj(conditional_past2/past/affirmative/informal)-->[-ra].

aux_suf_conj(predicative1/present/affirmative/informal)-->[].%predicative
form
aux_suf_conj(predicative2/present/affirmative/informal)-->[].

aux_suf_conj(continuative1/present/affirmative/informal)-->[-te].%continuat
ive form
aux_suf_conj(continuative2/present/affirmative/informal)-->[-de].

aux_suf_conj(conditional2/present/affirmative/informal)-->[-reba].%conditio
nal form
aux_suf_conj(conditional1/present/affirmative/informal)-->[-ba].

aux_suf_conj(imperative1/present/affirmative/informal)-->[].%imperative
form
aux_suf_conj(imperative2/present/affirmative/informal)-->[-ro].
aux_suf_conj(imperative3/present/affirmative/informal)-->[-i].

aux_suf_conj(negative_imperative1/present/negative/informal)-->[].%imperati
ve/negative form
```

```
aux_suf_conj(negative_imperative2/present/negative/informal) -->[].

aux_suf_conj(volitional1/present/affirmative/informal)-->[-o].%volitional
form
aux_suf_conj(volitional2/present/affirmative/informal)-->[-yoo].
```

APPENDIX 2: Outputs of the Consonant Verb *"kaku"*

[1] ?-verb('kaku',T,G,F,V,[]).

Soln: 1

T = to_write; G = cs; F = imperfective / present / negative / informal; V = [kak, - a, - na, - i]

;

Soln: 2

T = to_write; G = cs; F = imperfective / past / negative / informal; V = [kak, - a, - nakat, - ta]

;

Soln: 3

T = to_write; G = cs; F = imperfective / present / negative_continuative / informal;

V = [kak, - a, - naku, - te]

;

Soln: 4

T = to_write; G = cs; F = imperfective / present / negative_conditional / informal;

V = [kak, - a, - nake, - reba]

;

Soln: 5

T = to_write; G = cs; F = imperfective / past / negative_conditional / informal;

V = [kak, - a, - nakat, - ta, - ra]

;

Soln: 6

T = to_write; G = cs; F = causative2 / present / affirmative / informal; V = [kak, - a, - se, - ru]

;

Soln: 7

T = to_write; G = cs; F = passive2 / present / affirmative / informal; V = [kak, - a, - re, - ru]

;

Soln: 8

T = to_write; G = cs; F = polite / present / affirmative / formal; V = [kak, - i, - mas, - u]

;

Soln: 9

T = to_write; G = cs; F = polite / past / affirmative / formal; V = [kak, - i, - mashi, - ta]

;

Soln: 10

T = to_write; G = cs; F = polite / present / negative / formal; V = [kak, - i, - mase, - n]

;

Soln: 11

T = to_write; G = cs; F = polite / past / negative / formal; V = [kak, - i, - mase, - n, - deshita]

;

Soln: 12
T = to_write; G = cs; F = polite / past / affirmative_conditional / formal;
V = [kak, - i, - mashi, - ta, - ra]
;
Soln: 13
T = to_write; G = cs; F = polite / past / negative_conditional / formal;
V = [kak, - i, - mase, - n, - deshita, - ra]
;
Soln: 14
T = to_write; G = cs; F = polite / present / affirmative_volitional / formal;
V = [kak, - i, - masho, - o]
;
Soln: 15
T = to_write; G = cs; F = predicative1 / present / affirmative / informal; V = [kak, - u]
;
Soln: 16
T = to_write; G = cs; F = conditional1 / present / affirmative / informal; V = [kak, - e, - ba]
;
Soln: 17
T = to_write; G = cs; F = imperative1 / present / affirmative / informal; V = [kak, - e]
;
Soln: 18
T = to_write; G = cs; F = negative_imperative1 / present / negative / informal;
V = [kak, - u, - na]
;
Soln: 19
T = to_write; G = cs; F = volitional1 / present / affirmative / informal; V = [kak, - o, - o]
;
Soln: 20
T = to_write; G = cs; F = potential1 / present / affirmative / informal; V = [kak, - e, - ru]
;
Soln: 21
T = to_write; G = cs / process3; F = continuative1 / present / affirmative / informal; V = [kai, - te]
;
Soln: 22
T = to_write; G = cs / process3; F = past_tense1 / past / affirmative / informal; V = [kai, - ta]
;

Soln: 23

T = to_write; G = cs / process3; F = conditional_past1 / past / affirmative / informal;

V = [kai, - ta, - ra]

;

No More Solutions.

APPENDIX 3: Outputs of the Vowel Verb *"taberu"*

```
[1] ?-verb('taberu',T,G,F,V,[]).
Soln: 1
T = to_eat; G = vw; F = imperfective / present / negative / informal; V = [tabe, - na, - i]
;
Soln: 2
T = to_eat; G = vw; F = imperfective / past / negative / informal; V = [tabe, - nakat, - ta]
;
Soln: 3
T = to_eat; G = vw; F = imperfective / present / negative_continuative / informal;
V = [tabe, - naku, - te]
;
Soln: 4
T = to_eat; G = vw; F = imperfective / present / negative_conditional / informal;
V = [tabe, - nake, - reba]
;
Soln: 5
T = to_eat; G = vw; F = imperfective / past / negative_conditional / informal;
V = [tabe, - nakat, - ta, - ra]
;
Soln: 6
T = to_eat; G = vw; F = causative1 / present / affirmative / informal; V = [tabe, - sase, - ru]
;
Soln: 7
T = to_eat; G = vw; F = passive1 / present / affirmative / informal; V = [tabe, - rare, - ru]
;
Soln: 8
T = to_eat; G = vw; F = polite / present / affirmative / formal; V = [tabe, - mas, - u]
;
Soln: 9
T = to_eat; G = vw; F = polite / past / affirmative / formal; V = [tabe, - mashi, - ta]
;
Soln: 10
T = to_eat; G = vw; F = polite / present / negative / formal; V = [tabe, - mase, - n]
;
Soln: 11
T = to_eat; G = vw; F = polite / past / negative / formal; V = [tabe, - mase, - n, - deshita]
;
```

Soln: 12
T = to_eat; G = vw; F = polite / past / affirmative_conditional / formal;
V = [tabe, - mashi, - ta, - ra]
;
Soln: 13
T = to_eat; G = vw; F = polite / past / negative_conditional / formal
V = [tabe, - mase, - n, - deshita, - ra]
;
Soln: 14
T = to_eat; G = vw; F = polite / present / affirmative_volitional / formal; V = [tabe, - masho, - o]
;
Soln: 15
T = to_eat; G = vw; F = continuative1 / present / affirmative / informal; V = [tabe, - te]
;
Soln: 16
T = to_eat; G = vw; F = past_tense1 / past / affirmative / informal; V = [tabe, - ta]
;
Soln: 17
T = to_eat; G = vw; F = conditional_past1 / past / affirmative / informal; V = [tabe, - ta, - ra]
;
Soln: 18
T = to_eat; G = vw; F = predicative2 / present / affirmative / informal; V = [tabe, - ru]
;
Soln: 19
T = to_eat; G = vw; F = conditional2 / present / affirmative / informal; V = [tabe, - reba]
;
Soln: 20
T = to_eat; G = vw; F = imperative2 / present / affirmative / informal; V = [tabe, - ro]
;
Soln: 21
T = to_eat; G = vw; F = negative_imperative2 / present / negative / informal; V = [tabe, - runa]
;
Soln: 22
T = to_eat; G = vw; F = volitional2 / present / affirmative / informal; V = [tabe, - yoo]
;
Soln: 23
T = to_eat; G = vw; F = potential2 / present / affirmative / informal; V = [tabe, - rare, - ru]
;

Soln: 24

T = to_eat; G = vw; F = potential3 / present / affirmative / informal; V = [tabe, - re, - ru]

;

No More Solutions.

APPENDIX 4: Outputs of the Irregular Verb *"kuru"*

[1] ?-verb('kuru',T,G,F,V,[]).

Soln: 1

T = to_come; G = irreg / ko; F = imperfective / present / negative / informal; V = [ko, - na, - i]

;

Soln: 2

T = to_come; G = irreg / ko; F = imperfective / past / negative / informal; V = [ko, - nakat, - ta]

;

Soln: 3

T = to_come; G = irreg / ko; F = imperfective / present / negative_continuative / informal;

V = [ko, - naku, - te]

;

Soln: 4

T = to_come; G = irreg / ko; F = imperfective / present / negative_conditional / informal;

V = [ko, - nake, - reba]

;

Soln: 5

T = to_come; G = irreg / ko; F = imperfective / past / negative_conditional / informal;

V = [ko, - nakat, - ta, - ra]

;

Soln: 6

T = to_come; G = irreg / ko; F = causative1 / present / affirmative / informal;

V = [ko, - sase, - ru]

;

Soln: 7

T = to_come; G = irreg / ko; F = passive1 / present / affirmative / informal; V = [ko, - rare, - ru]

;

Soln: 8

T = to_come; G = irreg / ko; F = imperative3 / present / affirmative / informal; V = [ko, - i]

;

Soln: 9

T = to_come; G = irreg / ko; F = volitional2 / present / affirmative / informal; V = [ko, - yoo]

;

Soln: 10

T = to_come; G = irreg / ko; F = potential2 / present / affirmative / informal; V = [ko, - rare, - ru]

;

Soln: 11

T = to_come; G = irreg / ki; F = polite / present / affirmative / formal; V = [ki, - mas, - u]

;
Soln: 12
T = to_come; G = irreg / ki; F = polite / past / affirmative / formal; V = [ki, - mashi, - ta]
;
Soln: 13
T = to_come; G = irreg / ki; F = polite / present / negative / formal; V = [ki, - mase, - n]
;
Soln: 14
T = to_come; G = irreg / ki; F = polite / past / negative / formal; V = [ki, - mase, - n, - deshita]
;
Soln: 15
T = to_come; G = irreg / ki; F = polite / past / affirmative_conditional / formal;
V = [ki, - mashi, - ta, - ra]
;
Soln: 16
T = to_come; G = irreg / ki; F = polite / past / negative_conditional / formal;
V = [ki, - mase, - n, - deshita, - ra]
;
Soln: 17
T = to_come; G = irreg / ki; F = polite / present / affirmative_volitional / formal;
V = [ki, - masho, - o]
;
Soln: 18
T = to_come; G = irreg / ki; F = continuative1 / present / affirmative / informal; V = [ki, - te]
;
Soln: 19
T = to_come; G = irreg / ki; F = past_tense1 / past / affirmative / informal; V = [ki, - ta]
;
Soln: 20
T = to_come; G = irreg / ki; F = conditional_past1 / past / affirmative / informal; V = [ki, - ta, -
ra]
;
Soln: 21
T = to_come; G = irreg / ku; F = predicative2 / present / affirmative / informal; V = [ku, - ru]
;
Soln: 22
T = to_come; G = irreg / ku; F = conditional2 / present / affirmative / informal; V = [ku, - reba]
;

Soln: 23

T = to_come; G = irreg / ku; F = negative_imperative2 / present / negative / informal;

V = [ku, - runa]

;

No More Solutions.

APPENDIX 5: Outputs of the Irregular Verb *"suru"*

[1] ?-verb('suru',T,G,F,V,[]).

Soln: 1

T = to_do; G = irreg / sa; F = causative2 / present / affirmative / informal; V = [sa, - se, - ru]

;

Soln: 2

T = to_do; G = irreg / sa; F = passive2 / present / affirmative / informal; V = [sa, - re, - ru]

;

Soln: 3

T = to_do; G = irreg / sh; F = imperfective / present / negative / informal; V = [shi, - na, - i]

;

Soln: 4

T = to_do; G = irreg / sh; F = imperfective / past / negative / informal; V = [shi, - nakat, - ta]

;

Soln: 5

T = to_do; G = irreg / sh; F = imperfective / present / negative_continuative / informal;

V = [shi, - naku, - te]

;

Soln: 6

T = to_do; G = irreg / sh; F = imperfective / present / negative_conditional / informal;

V = [shi, - nake, - reba]

;

Soln: 7

T = to_do; G = irreg / sh; F = imperfective / past / negative_conditional / informal;

V = [shi, - nakat, - ta, - ra]

;

Soln: 8

T = to_do; G = irreg / sh; F = polite / present / affirmative / formal; V = [shi, - mas, - u]

;

Soln: 9

T = to_do; G = irreg / sh; F = polite / past / affirmative / formal; V = [shi, - mashi, - ta]

;

Soln: 10

T = to_do; G = irreg / sh; F = polite / present / negative / formal; V = [shi, - mase, - n]

;

Soln: 11

T = to_do; G = irreg / sh; F = polite / past / negative / formal; V = [shi, - mase, - n, - deshita]

;

Soln: 12
T = to_do; G = irreg / sh; F = polite / past / affirmative_conditional / formal;
V = [shi, - mashi, - ta, - ra]
;
Soln: 13
T = to_do; G = irreg / sh; F = polite / past / negative_conditional / formal;
V = [shi, - mase, - n, - deshita, - ra]
;
Soln: 14
T = to_do; G = irreg / sh; F = polite / present / affirmative_volitional / formal
V = [shi, - masho, - o]
;
Soln: 15
T = to_do; G = irreg / sh; F = continuative1 / present / affirmative / informal; V = [shi, - te]
;
Soln: 16
T = to_do; G = irreg / sh; F = past_tense1 / past / affirmative / informal; V = [shi, - ta]
;
Soln: 17
T = to_do; G = irreg / sh; F = conditional_past1 / past / affirmative / informal;
V = [shi, - ta, - ra]
;
Soln: 18
T = to_do; G = irreg / sh; F = imperative2 / present / affirmative / informal; V = [shi, - ro]
;
Soln: 19
T = to_do; G = irreg / sh; F = volitional2 / present / affirmative / informal; V = [shi, - yoo]
;
Soln: 20
T = to_do; G = irreg / su; F = predicative2 / present / affirmative / informal; V = [su, - ru]
;
Soln: 21
T = to_do; G = irreg / su; F = conditional2 / present / affirmative / informal; V = [su, - reba]
;
Soln: 22
T = to_do; G = irreg / su; F = negative_imperative2 / present / negative / informal; V = [su, -
runa]
;

Soln: 23

T = to_do; G = irreg / de; F = potential4 / present / affirmative / informal; V = [de, - ki, - ru]

;

No More Solutions.

CURRICULUM VITA

Mayumi Kobayashi was born in Japan. She has been fascinated with English since she visited Tokyo Disneyland in Japan at age 11. She was exposed to truly western oriented culture for the first time in her life. Since then, she has been cultivating English competence at school and at a private English conversation school. She graduated from Kawagoe High School in Japan majoring in English in the spring of March, 1994 and entered Ibaraki University. While pursuing a bachelor's degree in Education, she decided to study abroad at St. Cloud State University in St. Cloud, Minnesota from March 1997 to March 1998. She received her bachelor's of Education degree and the specialized English teaching certificate for elementary and middle schools from Ibaraki University in 1999. Afterwards, she started pursuing a career related English language. She has taught English in various settings. She has worked as an English instructor for adult second language learners in the private English conversation school in Japan and as a specialized English teacher for young children second langauge learners in the elementary and middle schools in Japan. Also, she has worked for a Japanese automobile supplier as a technical translator in Japan and the Philippines. In the fall of 2012, she entered the Graduate School at The Univeristy of Texas at El Paso.

Permanent Contact: mayumi.kobayashi930@gmail.com