Linguistic Project - Model Documentation (Feb 14, 2024 - Feb 23, 2024)

Introduction

Rukai is one of 16 traditional Taiwanese tribes, it consists of a variety of branches in terms of the spoken language, one of which is Budai Rukai.

The foot structure of Budai Rukai is a moraic trochee, where heavy syllables are more prominent and are parsed into its foot. The head of a prosodic word from Budai Rukai aligns with the right word edge, however, the last syllable is never parsed because the primary stress of a prosodic word is either on the penult or the antepenult. The exception to this observation appears when the word is monosyllabic, the syllable is parsed regardless. For bisyllabic words, only the first syllable is parsed and the last syllable is unparsed, regardless of their syllable weight.

Words with a monosyllabic suffix influence the position of primary stress to always be on the antepenult. Prefixes and infixes do not bear stress, only the root bears stress. Words with five or more syllables tend to carry secondary stress, the secondary stress is always on the left of the head. These observations are converted into a constraints ranking based on Optimality Theory.

Based on the observations and constraints, I wrote a Java program to model the expected foot parsing of a given lexical word from Budai Rukai. The code runs on top of Java 19 with the help of JavaFX for GUI support, but the base program can run as a console application with minor tweaking.

The sections below show how the program functions.

Intended Input & Output

The intended input for the program is the lexical word from Budai Rukai, specifically, the input should be in its morpheme form. The prefix, infix, root, and suffix should be separated with a hyphen, and a root indicator ‘[‘ should be placed before the root and after the hyphen.

The expected output should be the parsed version of the input according to the observations of the prosodic words of Budai Rukai.

Interpreting Phase

The interpreting phase is responsible for interpreting the lexical word. The result of the phase is a root location index (where the root of a word begins, if applicable) and the list of all syllables in the given word.

Identify Locations

```
private void identifyLocations() {
    for (int index = 0; index < this.input.length(); index++) {
        if (this.input.charAt(index) == '[') {
            this.rootLocation = index + 1;
        }

        for (char character : this.vowels) {
            if (character == this.input.charAt(index)) {
                this.vowelIndicies.add(index);
            }
        }
    }
}
```

Figure 1: The logic to identify the root location and the indices of the vowels in a word

According to Figure 1, the input (the given lexical word) is looped through. A string can be looped based on the length because it is a sequence of characters stored in an array. A representation of a string as a primitive type is char[] input = new char[size] (size is an arbitrary number), and each character in a text is added to the array in its sequence.

The for loop goes through each character in the input, if the character is a root indicator '[', then the root starts on the next character. The location, the index as an integer, is stored in a variable named "rootLocation".

```
public final char[] vowels = {'i', 'y', 'ɨ', 'ʉ', 'ɯ', 'u', 'ɪ', 'ʏ', 'ʊ', 'e', 'ø',

'ɘ', 'ɵ', 'ɤ', 'o', 'ə', 'ɛ', 'œ', 'ɜ', 'ɞ', 'ʌ', 'ɔ', 'æ', 'a', 'ɶ', 'ɑ', 'ɒ'};
```

Figure 2: The list of IPA vowels

The list of IPA vowels is created as a final character array, meaning it cannot change after initialization, as shown in Figure 2. The loop in Figure 1 also checks if the character

matches any vowels shown in Figure 2 and adds the index to an ArrayList named "vowelIndices".

Create Syllable List

With the acquired information on the vowel positions, the syllable list of the input word can now be created.

A StringBuilder object within the scope of the method is created to chain a series of characters to form a complete syllable. The method .append(char) chains the character to the existing StringBuilder and can be converted to a String through .toString() after the complete syllable is built.

The "vowelIndices" ArrayList is looped through and each vowel index is taken for the if statements shown in Figures 3, 4, and 5. Also, the StringBuilder uses the indices to form the complete syllable from getting the character at the index of the input word.

Then, the syllable is placed into an ArrayList named "syllables" for the parsing phase, and the StringBuilder is cleared for the next vowel through .setLength(0).

```
if (index - 2 >= 0 &&
    ((this.isLetter(this.input.charAt(index - 2)) &&
    this.isNotVowel(this.input.charAt(index - 2)) &&
    !this.isGlide(this.input.charAt(index - 2)) &&
    this.isNotVowel(this.input.charAt(index - 1))) ||
    (this.input.charAt(index - 2) == '-' && this.isLetter(this.input.charAt(index - 1))))) {
    syllable.append(this.input.charAt(index - 2));
}
```

Figure 3: The logic for the second character towards the left of the current vowel

According to Figure 3, the program checks for one of the following conditions:

1. If the second character towards the left of the current vowel is a letter and a consonant, not a glide, and the preceding character of the current vowel is a consonant.
2. If the second character towards the left is a section separator '-' and the succeeding character is a letter.

The character is built into the syllable if one of the conditions is true. The first condition ensures onsets with multiple consonants are available, and examples, such as "tsə," are accepted. The second condition ensures the section separator '-' is sorted into its syllable, this

reduces the parsing phase and compensates for situations where the section separator precedes a complete syllable, such as “-de”.

```
if (index - 1 >= 0 &&
    (this.isLetter(this.input.charAt(index - 1)) &&
    this.isNotVowel(this.input.charAt(index - 1)) ||
    this.input.charAt(index - 1) == '-')) {
    syllable.append(this.input.charAt(index - 1));
}
```

Figure 4: The logic for the preceding character of the current vowel

According to Figure 4, the program checks for the following conditions:

1. If the preceding character of the current vowel is a letter and is a consonant.
2. If the preceding character of the current vowel is a section separator ‘-’.

Similarly, the character is built into the syllable if one of the conditions is true. The first condition checks if the character preceding the vowel is the onset, which mostly only consists of one consonant, except for combining two consonants in the onset. The second condition checks if the preceding character is a section separator ‘-’ as this considers the case where the section separator is between the onset and the nucleus.

Then, the vowel itself is chained into the syllable.

```
if (index + 1 < this.input.length() &&
    (this.isGlide(this.input.charAt(index + 1)) ||
    this.isLongVowelIndicator(this.input.charAt(index + 1)))) {
    syllable.append(this.input.charAt(index + 1));
}
```

Figure 5: The logic for the succeeding character of the current vowel

According to Figure 5, the program checks for the following conditions:

1. If the succeeding character is a glide.
2. If the succeeding character is a long vowel indicator ‘:’.

The character is built into the syllable if one of the conditions is true. The first condition checks if a glide succeeds in the nucleus, such as “gaj”. The second condition checks if the succeeding character is a long vowel indicator ’:’, such as “ku:”. These checks are to ensure the heavy syllables are considered and separated properly.

Parsing Phase

The parsing phase is responsible for the foot parsing while following the observations of Budai Rukai while utilizing the “syllables” ArrayList created from the interpreting phase.

As discussed in ***Introduction***, the observations of Budai Rukai are converted into a constraints ranking according to Optimality Theory, the following sections are categorized based on the hierarchy of the constraints.

WDCOND

```
if (unparsedSyllables.size() == 1) {
    parsedWord.append("('").append(unparsedSyllables.get(0)).append(')');
    this.output = parsedWord.toString();
    return;
}
```

Figure 6: The logic to satisfy the constraint WDCOND

WDCOND requires the left of the foot to be aligned with the left word edge and the right of the foot to be aligned with the right word edge. This constraint ranks higher than NONFINALITY on monosyllabic words, meaning all monosyllabic words ought to be parsed into its foot.

The logic in Figure 6 simply checks if there is only one syllable in “unparsedSyllables”. The syllable will be parsed into its foot and be returned as the output.

HAVESTRESS/ROOT

```
for (String syllable : unparsedSyllables) {
    if (this.input.charAt(this.rootLocation) == syllable.charAt(0) &&
        this.input.charAt(this.rootLocation + syllable.length() - 1) == syllable.charAt(syllable.length() - 1)) {
        unparsedSyllables = unparsedSyllables.subList(unparsedSyllables.indexOf(syllable), unparsedSyllables.size());
        break;
    }
}
```

Figure 7: The logic to satisfy the constraint HAVESTRESS/ROOT

HAVESTRESS/ROOT requires stress to only be on the root of the word, disregarding the prefixes and the infixes. Similarly, this constraint ranks higher than NONFINALITY on words with prefixes and infixes, meaning only the word's root ought to be parsed and the ultima parsed if needed to satisfy TROCHEE.

As shown in Figure 7, the logic first checks if the root indicator '[' is present in the input. If the condition is true, the portion of the input until the root indicator '[' is added to "parsedWord". Then, the "syllables" ArrayList is looped through to find the syllable succeeding the root indicator. If the match is found, the portion of "syllables" starting from the matching syllable to the end of the list is taken and replaces "unparsedSyllable". The loop immediately breaks out after replacing "unparsedSyllable" to save process time.

NONFINALITY

```
if (unparsedSyllables.size() == 2) {
    if (!this.input.contains("[") || this.isHeavySyllable(unparsedSyllables.get(0))) {
        parsedWord.append("('").append(unparsedSyllables.get(0)).append(')').append(unparsedSyllables.get(1));
        this.output = parsedWord.toString();
        return;
    }

    parsedWord.append("('").append(unparsedSyllables.get(0)).append(unparsedSyllables.get(1)).append(')');
    this.output = parsedWord.toString();
    return;
}
```

```
for (int unparsedSyllableIndex = unparsedSyllables.size() - 2;
     unparsedSyllableIndex >= 0; unparsedSyllableIndex--) {
```

```
        parsedWord.insert(this.rootLocation, unparsedSyllables.get(unparsedSyllableIndex));
    }

    parsedWord.append(this.syllables.get(this.syllables.size() - 1));
    this.output = parsedWord.toString();
}
```

Figure 8: The logics that satisfy the constraint NONFINALITY

NONFINALITY requires foot parsing to begin at the penult or antepenult, which ranks higher than all constraints but WDCOND and HAVESTRESS/ROOT. This constraint forces the ultima to be unparsed, however, monosyllabic words or bisyllabic roots with preceding prefixes and infixes do not follow this constraint, and the ultima is, therefore, parsed.

There are two logics shown in Figure 8, the first is when there are only two syllables, and the second is a for loop. If the number of syllables is equal to 2, the first syllable is parsed and the second syllable is unparsed. The output is returned of the result is returned. If the

number of syllables is greater than 2, the “syllables” ArrayList is looped backward, beginning with the second to last item and appending the last syllable to the output. With each syllable, insert the syllable after any processes (if applicable) to “rootLocation” as the word before “rootLocation” has already been appended. The rest of the process is done within the for loop to maintain NONFINALITY and is parsed according to the satisfied constraints. The ultima is chained to the output when “syllables” are looped through and left unparsed.

WEIGHT-TO-STRESS

```
if (this.isHeavySyllable(unparsedSyllables.get(unparsedSyllableIndex)))
```

Figure 9: The logic to satisfy the constraint WEIGHT-TO-STRESS for heavy syllables

```
if (unparsedSyllableIndex - 1 >= 0 && this.isHeavySyllable(unparsedSyllables.get(unparsedSyllableIndex - 1))) {
    parsedWord.insert(this.rootLocation, unparsedSyllables.get(unparsedSyllableIndex));
    continue;
}
```

```
if (!unparsedLightSyllable.isEmpty())
```

```
if (unparsedSyllableIndex > 0)
```

Figure 10: The logic to satisfy the constraint WEIGHT-TO-STRESS for light syllables

WEIGHT-TO-STRESS requires each foot to be parsed according to the weight of the moras, this constraint ranks lower than NONFINALITY which means it must satisfy NONFINALITY and the constraints above it. WEIGHT-TO-STRESS requires a heavy syllable, containing two moras, to be parsed into its foot and a set of two light syllables, each containing one mora, to be parsed into its foot. Generally, the heavy syllable will be the head of the prosodic word and the primary stress is located at the penult. If no heavy syllables are present, the set of two light syllables (penult and antepenult) will be the head of the word and the primary stress will be at the antepenult.

According to Figure 9, the logic checks if the syllable is heavy. A heavy syllable includes the long vowel indicator ‘:’ or a glide ‘w’ or ‘j’. The if statement checks if the current syllable includes these characters, and parses the syllable into its foot. Then, the logic checks if there are only 4 syllables and leaves the remaining syllables unparsed. This check satisfies the observation that words with 4 or fewer syllables do not have secondary stress. The output returns the word if the condition is satisfied, otherwise, continue down the syllables.

According to Figure 10, the section of the code does various checks for the foot parsing of light syllables. In order, the condition makes sure the succeeding syllable is not parsed when the preceding syllable is heavy. The next few conditions are based on a local String named “unparsedLightSyllable,” this String will store the light syllable the “unpasredSyllableIndex” currently points to and continues down the loop. The third condition checks if “unparsedLightSyllable” is not empty and parses both syllables into their foot if the syllable preceding the one in “unparsedLightSyllable” is also light. The program breaks out the loop if it reaches the beginning item of “syllables”. Otherwise, the program continues after setting “unparsedLightSyllable” to an empty String. If “unparsedLightSyllable” is empty and the loop has not reached the beginning item of “syllables,” “unparsedLightSyllable” is set to the current item of “syllables.”

ALIGN-HD-R

ALIGN-HD-R requires the primary stress to be towards the right word edge, meaning all secondary stresses are towards the left of the primary stress. This constraint ranks lower than NONFINALITY and ranks the same as WEIGHT-TO-STRESS.

The program stores a local boolean variable named “primaryStressParsed,” once the heavy syllable or the penult and antepenult have been parsed, “primaryStressParsed” is set to true, and succeeding syllables are parsed with secondary stress. The primary stress of a given word, according to the IPA, is labeled with ‘',’ and the secondary stresses are labeled with ‘ˌ’.

GUI

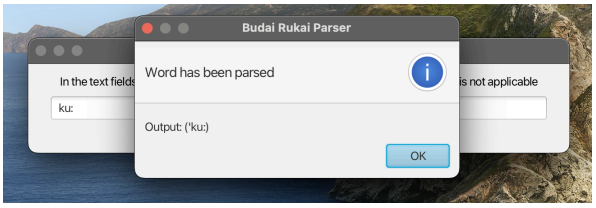
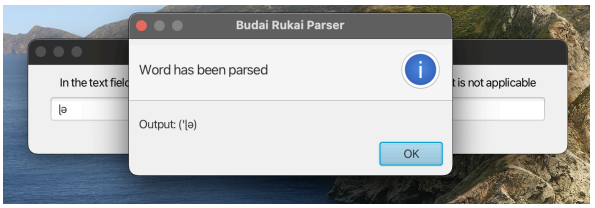
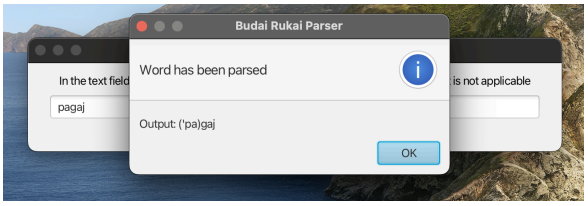
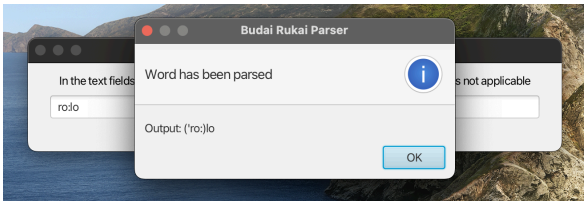
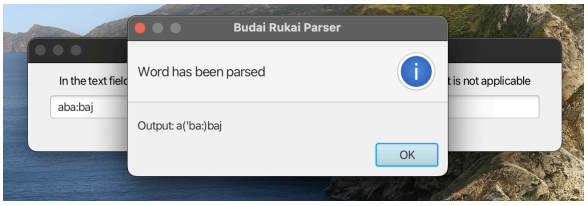
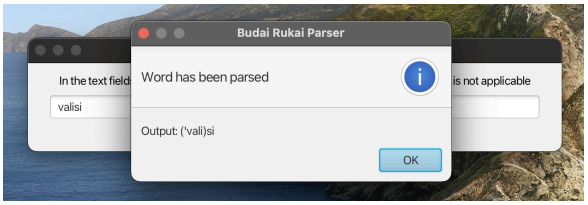
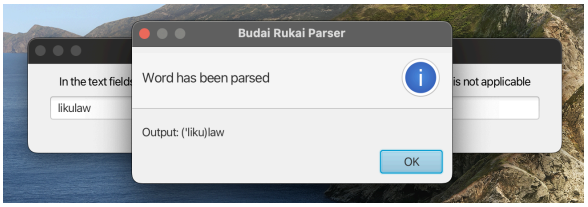
The GUI is handled with JavaFX, which makes it simpler for the user to input a lexical word. The program will prompt the user with a parsed version of the given word.

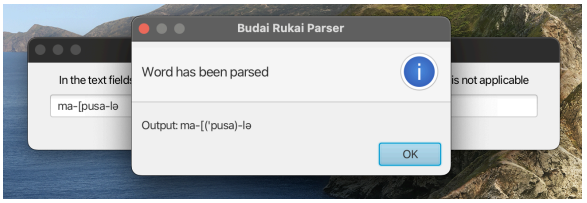
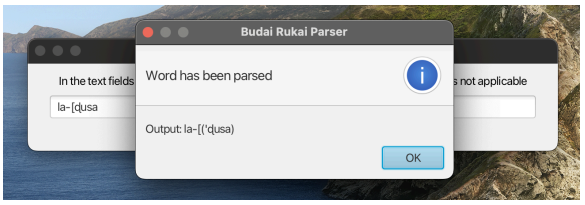
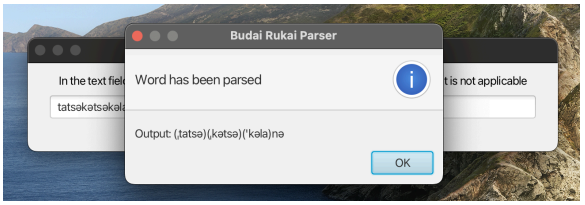
Example Inputs and Expected Outputs

This section demonstrates the

Table 1: The inputs and their corresponding expected outputs and actual outputs

Inputs	***Expected Outputs***	***Actual Outputs***

ku:	('ku:)	
ɭə	('ɭə)	
pagaj	('pa)gaj	
ro:lo	('ro:)lo	
aba:baj	a('ba:)baj	
valisi	('vali)si	
likulaw	('liku)law	

ma-[pusa-lə	ma-[('pusa)-lə	
la-[ɖusa	la-[('dusa)	
tatsəkətsəkəlanə	(ˌtatsə)(ˌkətsə)('kəla)nə	

Notes

The research for the foundation of the program is from Kuo-Chiao Lin's research on foot parsing research of Budai Rukai in 2014. The code for parsing a lexical word is written based on the constraints ranking from the paper. Table 1 in ***Expected Inputs and Expected Outputs*** shows the expected inputs and outputs, the expected inputs are provided in the "morphemes" section of the paper and the expected outputs are produced according to the constraints ranking of Budai Rukai from Lin's research.