ICS 661 Final Project Report

December 10, 2012

Jordan Do
Keone Hiraide

Introduction

Our project involved expanding assignments 3 and 4 which were assigned to us in the ICS 661 Fall 2012 semester. We created a grammar and lexicon for the Japanese language with respect to the language found in a Japanese comic book. Both authors of this paper have taken Japanese language classes in the past. As a result, we used our prior knowledge as well as various Japanese language websites as reference sources in creating our grammar and lexicon. Our assignment 4 program implemented the Japanese grammar and lexicon. We then used the very popular Japanese comic(Manga) series titled, “Hajime No Ippo” which was written by Morikawa Jouji as a data source. The original goal of our project was to be able to have our program recognize most passages contained within the manga as correct Japanese sentences, expressions, questions, and statements. After successfully identifying grammatical sentences, we decided to try make the project a little more interesting by translating those correctly structured Japanese sentences, expressions, questions, and statements into their English structured counterparts.

Description

For our project, we have implemented the Cocke–Younger–Kasami (CKY) algorithm in the C++ programming language. The CKY algorithm is a parsing algorithm that takes as input, Context Free Grammars (CFG). The parser only accepts grammars in Chomsky Normal Form (CNF) and so a converter was in made in Java to modify CFS to CNF. The CKY algorithm is used to determine whether an inputted text is a correctly structured sentence. Our program takes in Japanese Romaji, which are Japanese words written

using English characters. If the inputted text is a correctly structured Japanese sentence, our program would output to the console an “ S ” or “ Not S “. An output of “S” means that our program characterized the inputted as a correctly structured Japanese sentence. A “Not S” means that our program characterized the inputted text as an incorrectly structured Japanese sentence. If our program characterized the inputted sentence as a correctly structured Japanese sentence, a translation using Direct Translation from Japanese Romaji to English is done. In other words, we directly translate each word from Romaji to its counterpart in English. Ideally, we wanted to move and/or change the words around to flow better and fit the English structure better than just a 1:1 direct translation, but due to time constraints we were unable to do so.

Some of the issues that we encountered during our project was our non-mastery of the Japanese language. Although the authors have taken Japanese Language classes in the past, time has led to diminished a knowledge and understanding of the Japanese language. For example, some Japanese words, characters, sentence structures, etc. needed to be researched. Because we were getting our source words from a comic, it was also sometimes problematic because characters used colloquial language that was harder to look up (abbreviations of words or slang that dictionaries didn’t recognize). We discovered how Japanese and English have many differences both in their structure and in their lexicons, and this made translation somewhat difficult.

The second problem was creating the lexicon with regards to Japanese morphism. Verbs all have a root, and then depending on the form of the word, syllables are added

and sometimes some consonants change. We wanted to have the root word the same for each verb and then just look at the endings to determine the part of speech of the word, but that means each verb would contain two different tokens. This isn't so hard to write down in a lexicon, but when one types the romanji into the program, they have to be aware that words should be split occasionally. Another side effect of this, is that spelling errors sometimes became an issue. As romanji is an English equivalent to actual Japanese characters, we have seen different people use different spelling conventions and we had the choice of either including both variations or forcing a single one. For time concerns, we decided to go with a single style.

Finally, when translating to English, because of our style of lexicon (separated verbs) it meant that a direct translation wouldn't work as well as we would have liked. The word ikitai (split into "ik" and "itai" would come out directly as "go (want to)". Rather than doing a direct translation, it would have been nice to at least look at the two and change it to "want to go". In the same manner, "eat (past)" should convert to "ate". However, by having the direct translations, it is enough for someone who knows about our program's behavior to piece together sentences and translate it manually. Therefore, given a large enough lexicon, it is conceivable for someone who knows nothing about Japanese but how to read text would be able to use our program to help translate.

The only other problem was knowing which word to use when multiple distinct words have the same spelling. Because our program went through directly, it wasn't easy to use context to determine which word to use and so we either picked the first or

displayed them all.

Analysis

Despite the shortcomings of our translation, after looking at the output, it felt very similar to some dictionary/translators that we've used before. Things like Google Translate try to look for phrases and piece things together to sentences that make sense in English which is nice, but if you want to check for sure, it is often hard to tell what's being used where (if there are multiple readings etc). Besides Google, there are programs out there that do things very similar to our project (which is basically output definitions for words) and for people who are learning or know some Japanese (especially concerning structure and grammar) knowing what words mean and how endings change those words can be enough to help produce a good sentence because users can piece together the sentences themselves using what they know of grammar and the given vocabulary. So for people who know absolutely nothing, doing something like Google (which we originally wanted to do by moving words around and changing them) is probably the best plan but what our program does is not useless for people who already know a little.

As a small example, after running the program against the text in the comic book we used, we inserted the direct translations into the speech bubbles and the result was something almost readable that could be fixed up very easily.

Conclusion

We have learned more about the Japanese language after completing this project. Some areas that we gained knowledge on include the syntax, the meaning, and the morphology of the Japanese language.

If someone wanted to expand on our project, extending to our grammar and lexicon would be a great area to contribute to. Our translation algorithm could also be improved, extended, or replaced. We used the Direct Translation method as discussed in section 25.2.1 in Chapter 25 of the class textbook titled, “Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition”. If we tried other methods such as transfer and interlingua approaches, we may have gotten a more accurate and efficient translation mechanism.

Appendex A

```
#include <iostream>
#include <iomanip>
#include <fstream>
#include <map>
#include <sstream>
#include <cstring>
#include <algorithm>
#include <string>
#include <vector>
#include <cassert>
using namespace std;


string grammar[1000][1000]; //We store our grammar here from a text file. e.g. S->NP
VP
string parsing[1000]; //Used for parsing grammar
int lexicon_size, p;

int wordCount(string a); //counts the amount of words in a sentence
void stringHolder(string a, vector<string> &b); //Takes each word in a sentence and
places them in an index of a vector
string parser( string a, string b); //parses sentence and our grammar
void rightSide_Grabber(string a); //Looks at the grammar and seperates it on the right
side
string lookPossible(string a); //Looks at the possible grammars that can produce string a
string createPossible(string a, string b); //Creates all the possible grammars.
void readFile(string filename, vector<string> &data);//reads input file and populates a
vector
string translate(string sentence, string dictionary_file); //Direct translation using a
dictionary file from Japanese to English.

int main()
{
    int index,iter;
    string test,the_string,text,text2,start;
    vector<string> data;
    readFile("output.txt",data);
    start = "S";
    lexicon_size = data.size();

    for (int i=0; i < lexicon_size; i++)
    {

            test = data[i];
```

```
        index=test.find("->");
        grammar[i][0] = test.substr(0,index);

        test = test.substr(index+2, test.length());
        rightSide_Grabber(test);
        for (int j=0; j < p; j++)
        {
                grammar[i][j+1]=parsing[j];
        }
}


string line;
ifstream myfile ("sentences.txt");
ofstream ofile;
ofile.open ("example.txt");
if (myfile.is_open())
{
        while (true)
        {
                string storer[1000][1000],st;
                vector<string> holder;
                cout<<"\nPlease enter a Sentence" << endl;
                getline(cin,the_string,'\n');
                stringHolder(the_string, holder);
                for(int i=0; i < holder.size(); i++)
                {
                        text="";
                        st = "";
                        st=  st + holder[i];

                        for (int j=0; j < lexicon_size; j++)
                        {
                                iter=1;
                                while(grammar[j][iter].compare("") != 0)
                                {
                                        if(grammar[j][iter].compare(st) == 0)
                                        {
                                                text=parser(text,grammar[j][0]);
                                        }
                                        iter++;
                                }
                        }
                        storer[i][i]=text;
                }
```

```
                for (int k=1; k < holder.size(); k++)
                {
                        for (int j=k; j < holder.size(); j++)
                        {
                                text="";
                                for (int l=j-k; l < j; l++)
                                {
                                        text2 = createPossible(storer[j-k][l],storer[l+1][j]);
                                        text = parser(text,text2);
                                }
                                storer[j-k][j] = text;
                        }
                }

                for (int i=0; i < start.length(); i++)
                        if(storer[0][holder.size()-1].find(start[i]) <= storer[0][holder.size()-
1].length()) //Checks if last element of first row contains a Start variable
                        {
                                cout<<"S\n";
                                cout << translate(the_string,"translation.txt") << endl;
                                ofile << the_string << endl;
                                ofile << "S\n";
                                ofile << translate(the_string,"translation.txt") << endl;
                        }
                        else
                        {
                                cout<<"NOT S\n";
                                ofile << the_string << endl;
                                ofile << "NOT S\n";
                        }

        }
        myfile.close();
        ofile.close();
    }

    else {
        cout << "Unable to open file.. exiting" << endl;
        exit(0);
    }
}

int wordCount(string a)
//Counts the amount of words in a sentence.
{
    int count = 1;
```

```
    for (int i=0; i < a.size(); i++) {
        if (a[i] == ' ') {
            count++;
        }
    }
    return count;
}

void stringHolder(string a, vector<string> &b)
//Grabs a sentence and stores each word in a vector index
{
    string temp;
    for (int i=0; i < a.size(); i++) {
        temp = temp + a[i];
        if (a[i+1] == ' ' || i+1 == a.size()) {
            b.push_back(temp);
            i++;
            temp = "";
        }
    }
}

string parser(string a, string b)
//parses sentence and our grammar
{
    string temp=a;
    vector<string> word;
    stringHolder(b, word);
    for (int i=0; i < word.size(); i++) { //b should be changed to amount of words in right
side.
        temp = temp + word[i]; // word becomes temp
    }
    return temp;
}

void rightSide_Grabber(string a)
//Looks at the grammar and seperates it on the right side
{
    int i;
    p=0;

    while(a.length())
    {
        i=a.find("|");
        if(i>a.length())
        {
```

```
                parsing[p++] = a;
                a="";
        }
        else
        {
                parsing[p++] = a.substr(0,i);
                a=a.substr(i+1,a.length());
        }
    }
}

string lookPossible(string a) //returns a concatenated string of variables which can
produce string p
//Looks at the possible grammars that can produce string a
{
    int k;
    string temp="";
    for (int i=0; i < lexicon_size; i++)
    {
        k=1;
        while(grammar[i][k].compare("") != 0)
        {
                if(grammar[i][k].compare(a))
                {
                        temp=parser(temp,grammar[i][0]);

                }

                k++;
        }
    }
    return temp;
}

string createPossible(string a, string b)
//Creates all the possible grammars.
{
    string temp=a,newString="";
    vector<string> word1, word2;
    stringHolder(a,word1);
    stringHolder(b,word2);
    for (int i=0; i < word1.size(); i++)
    for (int j=0; j < word2.size(); j++)
    {
        temp="";
        temp=temp+word1[i]+word2[j];
```

```
        newString=newString+lookPossible(temp); //searches if the generated
productions can be created or not
    }
    return newString;
    }

void readFile(string filename, vector<string> &data)
//reads input file and populates a vector
{
    string line;
    ifstream myfile (filename.c_str());
    if (myfile.is_open())
    {
        while ( myfile.good() )
        {
            getline (myfile,line);
            data.push_back(line);
        }
        myfile.close();
    }

    else {
        cout << "Unable to open file.. exiting" << endl;
        exit(0);
    }
}

string translate(string sentence, string dictionary_file)
//Direct translation using a dictionary file from Japanese to English.
{
    ifstream myfile(dictionary_file.c_str());
    string line, substr, trans;
    map<string,string> data;
    map<string,string>::iterator it;
    size_t pos;
    if (myfile.is_open()) {
        while ( myfile.good() )
        {
            getline (myfile,line);
            pos = line.find(",");  // position of "live" in str
            string key = line.substr(0,pos);
            string value = line.substr(pos+1);
            data.insert ( pair<string,string>(key,value) );
        }
        myfile.close();
```

```
    }

    for (int i=0; i < sentence.size(); i++) {
        substr = substr + sentence[i];
        if (sentence[i+1] == ' ' || i+1 == sentence.size()) {
            i++;
            if (data.find(substr) != data.end()) {
                trans = trans + data[substr] + " ";
            }
            substr = "";
        }
    }

    return trans;

}
```

Appendex B

Our program uses as input, output.txt which contains our grammar and lexicon. Our grammar and lexicon is made up of text written in Romanji, which is Japanese words written using English characters. The user would input into our program a sentence. Our program would then indicate whether the inputted text is a proper Japanese sentence or phrase by outputting to the console, “S” or “Not S’”. An S means that our program characterizes the inputted text as a correctly structured Japanese sentence. A Not S means that our program charecterized the inputted text as an incorrect structured Japanese sentence. If the sentence inputted has been charecterized as a correctly structured sentence, out program would then translate that sentence from Japanese to English. translation.txt contains our dictionary. sentence.txt contains sample sentences that you can use as input for our program.

Sample Output:

Please enter a Sentence
eigami ni ik anee ka
S
movie theatre (by, through) go (negative) ?

Please enter a Sentence
ne sore watashi tachi mo mi tai
S
ne that i (multiple) also see (want to)

Please enter a Sentence
ne makunouchi mo ik ou yo
S
ne makunouchi also go (inviting) yo

Please enter a Sentence
kurasu gaes hita bakka de minna yoku shir anai doushi nan dakara
S
class start (past) only then everyone well know (negative) acquaintances ()
so/because/therefore

Bibliography

Hauck's Japanese Translation Service. (2004-2012). *Free Japanese Lessons.* Retrieved from http://www.freejapaneselessons.com/

Learn Japanese Language. (2011). *Learn Japanese Language: Learn all about Japanese's language from basic to advance, Basic Grammar, Conversation Sample.* Retrieved from http://japanese-tutorial.blogspot.com/

Let's Learn Japanese. (2010). *Sentence-final particles : yo, ne, no, sa, zo, ze, na and wa.* Retrieved from http://www.facebook.com/note.php?note_id=116422641715796

RawManga|Download Free Manga. (2012). *RawManga|Download Free Manga: Manga is best uncooked.* Retrieved from http://www.rawmangaspot.com/

Shinji, T. (n.d.) *Teach Yourself Japanese.* Retrieved from http://www.sf.airnet.ne.jp/~ts/japanese/