Robert Fiszer

ECE 578

**MARION WELSH**

MARION WELSH is a text comprehension program that behaves as the inverse of a Backus-Naur Form sentence generator. A BNF sentence generator will create a sentence that is syntactically valid, but does not necessarily make any semantical sense. MARION WELSH does the opposite, taking a supposedly syntactically and semantically valid sentence and decomposing it into its constituent parts of speech and combining the words into phrases until a single rule is reached. In her current state, MARION WELSH can decompose any sentence that can be modeled by a grammar specified by Robert Fiszer.

The grammar used follows a strict pattern. Each phrase consists of one, two or three sub-units. Each subunit can be a word (terminal case) or another phrase (non-terminal case). By repeatedly combining the words and phrases into a single tree, MARION WELSH can then use the syntactical hierarchy or tree to comprehend the passed sentence. This feature has not yet been implemented beyond a simple examination that identifies the type of sentence that is passed to MARION WELSH.

MARION WELSH is an acronym that stands for My Acronym Really Is Outrageously Neurotic Wise English Language Syntactical Hierarchy. In keeping with tradition, this natural language program is giving a female name. Furthermore, despite having a last name, this name is not a reference to any real person, dead or alive.

MARION is intended to be used with external files which allows for large amounts of information, while keeping the program itself as lightweight and readable as possible. The current standard forms are simple and the information is stored in text files, but this does not necessarily represent a finalized form. Ideally, the program would work with an SQL style database eliminating the startup times which could become very long once a large amount of words are defined.

The primary benefit of MARION compared to ELIZA, the previous program used for this purpose on the guidebot, is that ELIZA needed a database of regular expressions that she would try to match against the inputted sentence. MARION provides several advantages and disadvantages over this approach. ELIZA has the advantage in terms of guaranteeing sentence accuracy. ELIZA’s output will be prepared sentences to anticipated sentences. However, this is also ELIZA’s major flaw. ELIZA will be completely unable to understand a sentence that has been predicted, but in an unpredicted form. MARION’s advantage is that she will recognize any sentence that contains words that she knows in a syntax that she has been taught.

**SOURCE CODE**

The client would begin by initializing a MARION instance. This makes MARION initialize mappings from external files (txt files in a specified form). After the client has a MARION object, the client can call the comprehend method on a String. The method converts the sentence in a list of Syntactical Units and does a few error checking steps, and then does the actual work.

MARION calls a recursive method on the sentence. This method starts at the first entry in the list and tries to find a rule that matches the Syntactical Units in the list. When a match is found, MARION removes the matching units from the list, puts them in a new phrase object, and inserts the phrase into the list. MARION repeatedly performs this on the first entry, until she fails to make a match, then she skips forward one space, and repeats this process. However, once MARION is in the skip forward state, a match causes her to backtrack instead of trying again in the same spot. If MARION doesn’t find a match within two steps while backtracking, MARION continues stepping forward. MARION continues this pattern until either she steps off the end of the list or finds enough matches to reduce the list to a single tree.

**MARION.java**

**import** java.io.\*;

**import** java.util.\*;

**public** **class** Marion {

**private** Map<String, String> pos;

//private Map<String, String> definitions;

**private** Map<String, String[]> grammar;

**public** Marion() {

Scanner scan = **null**;

String str;

pos = **new** HashMap<String, String>();

grammar = **new** TreeMap<String, String[]>();

//initialize parts of speech mappings (go is a verb, bike is a noun, etc.)

**try** {

scan = **new** Scanner(**new** File("C:\\pos.txt"));

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**while**(scan.hasNext()) {

str = scan.next();

pos.put(str, scan.nextLine().trim());

}

scan.close();

//section to scan in definitions. This will be replaced with MySQL

//try {

// scan = new Scanner(new File("C:\\define.txt"));

//} catch (FileNotFoundException e) {

// e.printStackTrace();

//}

//while(scan.hasNext()) {

// str = scan.next();

// definitions.put(str, scan.nextLine());

//}

//scan.close();

//scan in grammar rules (a sentence is a nounphrase followed by a verbphrase, etc.)

**try** {

scan = **new** Scanner(**new** File("C:\\grammar.txt"));

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**while**(scan.hasNext()) {

str = scan.next();

grammar.put(str, scan.nextLine().trim().split("\\|"));

}

scan.close();

}

//creates a list of SUnits (syntactical units)

//words are single nodes, phrases are trees with words as leaf nodes

**public** **void** comprehend(String sentence) {

Scanner scan = **new** Scanner(sentence);

List<SUnit> list = **new** LinkedList<SUnit>();

String name;

**while**(scan.hasNext()) {

name = scan.next();

SUnit neo = **new** Word(name, pos.get(name)); //creates word objects

**if**(neo.type==**null**) {

wordError(name); //when present with a word that it does not know, displays info about the unknown word (not an exception)

**return**;

}

list.add(neo);

}

scan.close();

**boolean** status = match(list, 0, 0);

**if**(status&&list.get(0).type.equals("verbphrase")) { //implicitly makes a command

list.add(0, **new** Word("Marion", "marion"));

status = match(list, 0, 0);

}

**if**(status){

understand((Phrase) list.get(0));//if the program understands the sentence

} **else** {

grammarError(); //if it doesn't

}

}

**private** **void** understand(Phrase phrase) { //very minimal will need to be vastly expanded

//for this to actually understand meaning rather than just structure

System.*out*.println("You sentence is a "+ phrase.type+".");

**if**(phrase.type.equals("sentence")) {

System.*out*.println("This is an indicative statement. I'm going to ignore this.");

}**else** **if**(phrase.type.equals("command")) {

System.*out*.println("This is a command. " +

"I will look up the verb in my database (once I have one) and perform the related function." +

"\nThe DO and/or IO will be used as parameters");

}

}

**private** **void** wordError(String name) { //word not in lexicon

System.*out*.println("I'm sorry, I don't understand what "+name+" means.");

}

**private** **void** grammarError() {//the program cannot create a sentence out of the given words

System.*out*.println("I'm sorry, I understand the words, but not the sequence that you used them in.");

}

**private** **boolean** match(List<SUnit> list, **int** start, **int** status) { //the brains of the program

**if**(list.size() == 1) {//case when entire sentence is reduced to one phrase

**return** **true**;

}

//out of bounds checks

**if**(start <0) {

**return** **false**;

}

**if** (start >= list.size()) {

**return** **false**;

}

**for**(String s: grammar.keySet()) {//cycle through every phrase

**for**(String rules: grammar.get(s)) {//cycle through every rule for a phrase

String[] targets = rules.split(" "); //convert rules into a usable form

**if**(targets.length > list.size()-start) {//check if sentence is longer than rule

**continue**; //if so, go to next rule, or next phrase if at last rule for a phrase

}

**int** i = start; //where in the list we are looking

SUnit current;

**boolean** hack=**false**;

**for**(String type: targets) {//go though the rule, see if the next elements in the list match the rule

current = list.get(i);

**if**(!current.type.equals(type)) {

hack=**true**;

**break**;//if not a match, go to next rule or phrase

}

i++;

}

**if** (hack) {

**continue**;

}

//a match was found!

SUnit test = **new** Phrase(s); //create a phrase

**for** (**int** j = start; j < i; j++) {

((Phrase) test).add(list.remove(start)); //remove words/phrases from the list and put them in the phrase

}

list.add(start, test); //add the phrase to the place where we started at

**boolean** result;

**if** (status==0) {

result = match(list, start, 0);

}**else** **if**(status==1) {

result = match(list, start-1, -1);

}**else** **if**(start!=0){//backtracking and not at beginning

result = match(list, start-1, -1);

} **else**{//backtracking and at beginning

result = match(list, start, 0);

}

**if** (result) {//got a match

**return** **true**;

}**else** {

undo(list, start);

}

}

}

//no match found

**if**(status == -1) {

**return** match(list, start-1, -2);

}**else** **if**(status == -2) {//backtracking, no change therefore no change previously either

**return** **false**;

}**else**{

**return** match(list, start+1, 1);

}

}

**private** **void** undo(List<SUnit> list, **int** start) {

Phrase current = (Phrase) list.remove(start);

**for**(SUnit link: current.links) {

list.add(start, link);

start++;

}

}

}

**SUnit.java**

SUnit is the base class for Word and Phrase. Although java conventions state that I should make this an interface rather than a base class, it is slightly more efficient to make it a base class rather than an interface. It holds the two fields that Word and Phrase have in common.

**public** **class** SUnit {

**public** SUnit next;

**public** String type;

}

**Word.java**

Word is a class that holds word objects. Each word object stores the name of the word as well as the Part of Speech that the word is.

**public** **class** Word **extends** SUnit{

**public** String name;

**public** Word(String name, String type) {

**this**.name = name;

**this**.type = type;

}

}

**Phrase.java**

The class that holds phrase objects. Phrase objects have a list of links to their daughter objects.

**import** java.util.\*;

**public** **class** Phrase **extends** SUnit {

**public** List<SUnit> links;

**public** Phrase(String s) {

**this**.type = s;

links = **new** LinkedList<SUnit>();

}

**public** **void** add(SUnit next) {

links.add(next);

}

}

**MarionTest.java**

This is simply a method to test MARION and do things with her while not integrated with the guidebot proper. It does nothing more than initialize MARION and pass her a user-defined sentence.

**import** java.util.Scanner;

**public** **class** MarionTest {

**public** **static** **void** main(String[] args) {

Marion marion = **new** Marion();

**int** choice = -1;

Scanner scan = **new** Scanner(System.*in*);

**do** {

System.*out*.println("1. Enter a sentence\t\t0. Quit");

choice = Integer.*parseInt*(scan.nextLine());

**if**(choice == 1) {

System.*out*.println("Type in the sentence");

marion.comprehend(scan.nextLine());

}

} **while**(choice!=0);

}

}

**CONCLUSION**

MARION doesn’t have any semantical processing implemented yet, but she correctly parsed every simple statement and command that I added to the lexicon and grammar. MARION did this with nothing more than knowing the patterns that sentences come in and the part of speech that each word was. MARION requires still much more work to serve her primary purpose, but in her current state, she has a stable syntax engine which can be used by any future group looking to expand on this concept.