Christopher Foo

ICS 668

Dr. David Chin

Generating Language from Neo-Davidsonian Semantic Forms

Introduction

Handling meaning natural languages, while a very complex problem, can essentially be broken down into two broad issues. On one hand, systems need to be able to understand natural language sentences which typically involve breaking those sentences into more abstract semantic representations. On the other hand, those semantic representations also need to be translated back into natural language sentences for them to be suitable for consumption by the average human. This system attempts to tackle the latter problem of language generation in the limited domain of "looking for" sentences in massively multiplayer online games (i.e. looking for groups, looking for more members for groups, and buying and selling items).

Overall, the main goal of this system is to generate sentences based on a user provided semantic representation in the form of a text string. Consequently, two sub-goals emerge. Of course, one sub-goal is the actual generation of the sentences based on the information conveyed the text string. However, the information from the string must be in a system-recognizable form before that may happen. Thus, the second sub-goal is to parse the semantic representation into a form that the system can then use to accomplish the ultimate goal of generating the natural language sentences.

Description

The current version of the language generation system was created using Java 1.6 without any external parsing or language generation libraries. As a result, both the parser used to process the given semantic representation and the generator used to create the sentences was written by hand. The resulting system contains 19 Java classes with 7 implementing the parser and the other 12 implementing the language generator. Sample inputs and outputs may be seen in Appendix A and the source code may be seen in Appendix B.

The first problem encountered while making the system was deciding how to represent the semantics of the sentence. There are several predicate-based methods of representing semantics, but most are very difficult to parse due to the varying numbers and positions of arguments in the predicates. Fortunately, neo-Davidsonian representations resolve this issue by breaking single events into multiple predicates that are related to each other by lowercased variables (Jurafsky & Martin 566). This retains the representational ability of the other forms while reducing the predicates to just three forms (a single variable argument event predicate, a two variable argument "predicate, and a single variable argument, single non-variable argument "attribute" predicate). However, this comes at the cost of making the representations more verbose. Even so, the easier parsing of neo-Davidsonian representations outweighs the cost of longer input strings and hence was chosen as the semantic representation form for this system.

Representing neo-Davidsonian representations as text strings was another problem. Basic first order logic operators like ∧, ∨, and ¬ are difficult to enter into a standard ASCII text string and were therefore substituted with other more common characters (∧ = ,, ∨ = ;, and ¬ = -). Furthermore, the use of lowercased strings as variables prevents lowercased values from being used as attributes. This was alleviated by designating double and single quoted strings as non-variables as well as capitalized strings. Finally, the existential quantifier for each

variable was removed as it would invariably be the same when representing simple sentences like the ones handled by this language generator.

The parser works by finding the end of each predicated, signaled by a closing parenthesis, and checking the characters in the expected positions. For example, the logic operation is determined by looking at the first non-whitespace character and determining if it is a comma, semi-colon, or minus sign; the predicate argument list is the string between the opening and closing parenthesis; and the predicate type is the string before the opening parenthesis and after the comma, semicolon, or minus sign if one is present. This design was chosen because it was how I logically parsed the predicates, but it is easy to break with incorrect input strings. Nevertheless, it works correctly when a syntactically valid string is given to the program and successfully stores the predicates' information in the ParseToken objects that are used in the next step.

While attribute predicates can be properly connected to the predicates they are modifying (their "parents") in the linear list by matching their variables, it is inefficient as the full list must be searched to find them. As a result, the ParseTokens are arranged in a Tree structure that directly links the attribute predicates with their parents to limit the search space when trying to find all of the attributes of a predicate. Tree traversal can then be used to generate the objects needed in the generation step.

Now that the representation has been fully parsed, the generator can finally use the information stored in the generated objects to create the sentences. In essence, the generator uses context free grammars (Jurafsky & Martin 387) to generate the sentences by utilizing the overridden toString methods of the objects in the sentence and entity packages (besides the abstract Entity and Sentence classes). This method was chosen as creating a string

representation of a Java object using its toString method is analogous to resolving the rules for that object's non-terminal in a context free grammar. A complete listing of the context free grammar rules used to generate the sentences in Extended Backus-Naur Form (Reed) maybe seen in Appendix C.

The grammar rules used in this system can be broken down into three main categories: sentence rules, list rules, and Entity rules. Sentence rules are the top level rules that specify the structure of a particular type of sentence and they typically consist of lists generated by the list rules. The list rules are relatively simple, but list generation can be difficult if the logic operation between each of the elements is not consistent. As a result, the language generator does not use single lists. Instead, it uses lists of lists so that the outer list can contain only ∨ operations while the inner lists can only use ∧ operations to avoid the aforementioned problem. Finally, there is the Entity rules used to resolve the non-terminals left behind by the list rules. Here, the parsed information is used to create a completed string that will resolve the non-terminals from the previous expansions and ultimately create the sentence that will be printed to the screen.

Analysis

Overall, this language generator works well when given a syntactically correct string as it can parse any predicate in neo-Davidsonian form provided that all parent predicates (those with other predicates linked to them with a variable) are defined before their children (the predicates that are attached to them). Unrecognized predicates are ignored by the generation system which allows prevents it from crashing or creating strange outputs if an unrecognized predicate is encountered. Nevertheless, unrecognized predicates are parsed and can be seen if the user runs the program with the "–t" flag to print the parse trees. The generator even allows for multiple

sentences to be defined in a single representation as each would be parsed into its own tree and all trees are passed to the generator. Therefore, it does accomplish the main goals of parsing a neo-Davidsonian representation and generating a sentence based on the semantics of the parsed input.

Yet, the language generator is still limited in some aspects. The number of predicates recognized by the system is rather small with only four sentence predicates and four Entity predicates (an Entity is essentially a thing or noun). Additionally, the types of the subjects and objects for each sentence category are limited to certain values to ensure that the sentences are logically sound. In the current version, there is at most one valid object type and one valid subject type for each category which prevents many otherwise legitimate sentences from being generated. For example, the object in a BuySentence can only be an Item at the moment which prevents the user from generating sentences about buying other things such as a dungeon run. The generator also does not support negation. The parser properly stores any negation encountered in the predicate string, but the generator does not use that information to modify the generated sentences. This is in part due to the structuring of the semantic information when the tree is converted into the Java objects used in the generation step. In its current form, the object and subject predicates of the Sentence object generated are consolidated into a single object and a single subject which loses some of the negation information from gleaned the parse. As a result, a restructuring of the Sentence objects would be necessary to fully support negation in addition to changes in the grammar rules used to generate the final sentences.

Moreover, some of the software design choices are not optimal. As mentioned in the description section, the parser parses the predicates based on character position. While this does work, a better alternative would be to use regular expressions to create a more robust parsing

system. I also planned to use more reflection in the object generation step to make the system easier to extend, but most of the objects are generated by parsing the trees using hardcoded methods. An improvement would be to use reflection to determine which classes and fields are available so that adding a new Entity, Sentence, or attribute would be as simple as adding a few new classes to the appropriate packages and possibly adding a couple of lines to an object's toString method. Consequently, there is much room for improvement despite the successful completion of the system's goals.

Conclusion

In conclusion, the language generator implemented for this project does achieve its initial goals of parsing a semantic representation in a slightly modified neo-Davidsonian form and using context free grammars to turn the parsed information into a coherent English sentence. Of course, there are some short comings due to the limited nature of the program and it does not quite cover the full domain of "looking for" sentences in MMORPGs as a result. Furthermore, work on this project has revealed just how complicated natural languages are as the number of possible semantic combinations, even in a small of a domain like the one targeted in here, is quite large. This does not even touch the possibility of valid syntactic variations. However, there appeared to be some similarities between the parsing of each Entity type and the generation of each Sentence category as evinced in the amount of shared code in the current version (I also ended up doing a fair amount copy and pasting because I could not figure out how to generalize the code using reflection). As a result, any future attempts at this style of language generation should probably start with a smaller domain and fully exhaust it before expanding. By doing so, most of the parsing and generation code will have been completed and extensions can be easily

created by reusing the existing code and adding the necessary exceptions and peculiarities of the new domain. Nevertheless, the current version serves as a good starting point as the base parsing and generation systems work and the rest of the target domain can be covered by extending the generation capabilities of the existing system.

References

Jurafsky, D., & Martin, J. H. (2009). *Speech and language processing, an introduction to natural language processing, computational linguistics, and speech recognition*. (2nd ed.). Upper Saddle River, NJ: Prentice Hall.

Reed, N. (n.d.). *ICS313 Programming Language Syntax*. Retrieved from http://www2.hawaii.edu/~nreed/ics313/lectures/02syntax6up.pdf

Appendix A: Sample Runs

Note: Inputs are in `blue`

A.1 BuySentence

```
buy(a)
< Error in BuySentence: No object to buy found >

buy(a),object(a,obj),item(obj, item),name(item, "Thunderfury"),quantity(item, 2)
WTB 2 Thunderfuries.

buy(a),object(a, obj),item(obj,item1),item(obj,item2),item(obj,item3),
name(item1, 'Linen Cloth'),quantity(item1,4),name(item2,'Wool Cloth'),
quantity(item2, 10), name(item3, 'Silk Cloth'),quantity(item3, 500),
value(item3, value), moneyamount(value,amount),denomination(amount,2000),
currency(amount,'gold')
WTB 4 Linen Cloths, 10 Wool Cloths, and 500 Silk Cloths for 2.0k gold.

buy(a),subject(a, subj),player(subj,play),class(play,Shaman),
itemlevel(play, 200),object(a,obj),item(obj,item1),type(item1, Sword),
rarity(item1,Epic);object(a,obj2),item(obj2, item2),type(item2, Dagger),
quantity(item2, 2)
Item Level 200 Shaman WTB 1 epic sword or 2 daggers.
```

A.2 SellSentence

```
sell(a),subject(a,subj),player(subj,play),level(play,20),role(play,'tank'),
object(a,obj),item(obj,item1),name(item1,Great Sword),value(item1,val1),
moneyamount(val1,amount1),denomination(amount1,10),
currency(amount1,'gold'),moneyamount(val1,amount2),
denomination(amount2, 50),currency(amount2,'silver');item(obj,item2),
name(item2, Broken Shield)
Level 20 tank WTS 1 Great Sword for 10.0 gold and 50.0 silver or 1 Broken Shield.

sell(a),subject(a,subj),player(subj,play),level(play,20),role(play,'tank'),
object(a,obj),item(obj,item1),name(item1,Great Sword),value(item1,val1),
moneyamount(val1,amount1),denomination(amount1,10),currency(amount1,'gold'),
moneyamount(val1,amount2),denomination(amount2, 50),currency(amount2,'silver');
item(obj,item2),name(item2, Broken Shield),value(item2, val2),
moneyamount(val2, amount3),denomination(amount3,10),currency(amount3,'copper')
Level 20 tank WTS 1 Great Sword for 10.0 gold and 50.0 silver or 1 Broken Shield for
10.0 copper.
```

A.3 FindGroupSentence

```
findgroup(a),subject(a,subj),player(subj,play),class(play, Priest),class(play, Mage)
Priest / Mage LFG.

findgroup(a),object(a,obj),instance(obj,inst1),name(inst1,BFD);instance(obj,inst2),na
me(inst2,HoR),difficulty(inst2,'heroic')
LFG for BFD or heroic HoR.
```

```
findgroup(a),subject(a,subj),player(subj,play1),class(play1,
Warrior),specialization(play1, Protection),specialization(play1,
Fury),player(subj,play2),role(play2,'healer'),object(a,obj),instance(obj,inst1),name(
inst1,BFD);instance(obj,inst2),name(inst2,HoR),difficulty(inst2,'heroic')
Protection / Fury Warrior and healer LFG for BFD or heroic HoR.
```

A.4 FindMoreSentence

```
findmore(a),instance(a,inst),name(inst,'Deadmines'),difficulty(inst,'normal'),
mode(inst,'10 man)
LFM for normal 10 man Deadmines.

findmore(a),object(a,obj),player(obj,play1),role(play1, 'healer'),quantity(play1,
2),player(obj,play2),class(play2,Rogue),specialization(play2,Assassination)
LF3M 2 healers and Assassination Rogue.

findmore(a),object(a,obj),player(obj,play1),role(play1, 'healer'),quantity(play1,
2);player(obj,play2),class(play2,Rogue),specialization(play2,Assassination)
LF1-2M 2 healers or Assassination Rogue.
```

A.5 Multiple Sentence

```
Buy(a), ContactMethod(a, "PST"); ContactMethod(a, "find me"), Subject(a, sub),
Player(sub, play1), class(play1, Shaman), specialization(play1, Restoration),
level(play1, 85), Object(a,b), Item(b,c), Name(c,Thunderfury), Quantity(c,2),
Item(b,d),Type(d,Sword),Quantity(d,1),Item(b,h),Type(h,Shield),Quantity(h,10),
Value(h,i),MoneyAmount(i,j),Denomination(j,100),Currency(j,Gold),MoneyAmount(i,k),
Denomination(k, 20),Currency(k, Silver);-Item(b,item),Type(item,Bow),Sell(e),
Object(e,f),Item(f,g), Name(g,Wool Cloth),Quantity(g,5),FindGroup(fg),
Subject(fg,subFg),Player(subFg,fgPlay),Class(fgPlay,Shaman),Level(fgPlay,80);
Player(subFg,fgPlay2),Class(fgPlay2,Paladin),Level(fgPlay2,80),
Object(fg, fgObj),Instance(fgObj,fgInst1),Name(fgInst1, DS),
Mode(fgInst1,25 man),Difficulty(fgInst1,Normal),FindMore(fm),instance(fm,fmInst1),
name(fmInst1,'Mogushan Vaults'),object(fm,fmOb1),player(fmOb1,fmPlay1),
class(fmPlay1,Shaman), quantity(fmPlay1,2),player(fmOb1,fmPlay3),role(fmPlay3,Tank),
quantity(fmPlay3,2);player(fmOb1, fmPlay2),class(fmPlay2,Paladin),
quantity(fmPlay2, 3),instance(fm,fmInst2),
name(fmInst2,'Terrace of Endless Spring');instance(fm, fmInst3),
name(fmInst3,'Heart of Fear')
Level 85 Restoration Shaman WTB 2 Thunderfuries, 1 sword, and 10 shields for 100.0
gold and 20.0 silver or 1 bow, PST or find me.
WTS 5 Wool Cloths.
Level 80 Shaman or Level 80 Paladin LFG for Normal 25 man DS.
LF3-4M 2 Shamans and 2 Tanks or 3 Paladins for Mogushan Vaults and Terrace of Endless
Spring or Heart of Fear.
```

Appendix B: Source Code

B.1: edu.hawaii.ctfoo.lang_generator

B.1.1 Generator.java

```java
package edu.hawaii.ctfoo.lang_generator;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.sentence.Sentence;

/**
 * Reads in neo-Davidsonian like semantic representations from STDIN and prints
 * out the sentence forms of those representations. Also provides some String
 * manipulation function to be used in generating the sentences.
 *
 * @author Christopher Foo
 *
 */
public class Generator {

    /**
     * The {@link Parser} object used to parse the input from STDIN.
     */
    private Parser parser;

    /**
     * A {@link BufferedReader} to STDIN used to read in the representations.
     */
    private BufferedReader in;

    /**
     * Creates a new Generator and initializes its fields.
     */
    public Generator() {
        this.parser = new Parser();
        this.in = new BufferedReader(new InputStreamReader(System.in));
    }

    /**
     * Closes the input stream.
     */
    public void closeStream() {
        try {
            this.in.close();
        } catch (IOException e) {
            System.err.println("Error: Could not close input stream.");
        }
    }

    /**
     * Attempts to change the given string to it's plural form.
     *
     * @param string
```

```
     *            The String that should be made into its plural form.
     * @return The plural form of the given string.
     */
    public static String pluralize(String string) {
        char lastChar = Character
                .toLowerCase(string.charAt(string.length() - 1));
        if (lastChar == 'y') {
            return string.substring(0, string.length() - 1) + "ies";
        } else if (lastChar == 's') {
            return string + "es";
        } else {
            return string + "s";
        }
    }

    /**
     * Appends the given list to the given builder as a comma separated list
     * terminated by the given terminator string.
     *
     * @param objects
     *            The list of objects to be appended.
     * @param builder
     *            The {@link StringBuilder} to append the created list to.
     * @param terminator
     *            The terminator string of the entire list.
     * @param innerListTerminator
     *            The terminator strings of the sublists. The first will be used
     *            to terminate the first level sublists, the second the second
     *            level sublists, etc.
     */
    public static void appendCommaList(List<?> objects, StringBuilder builder,
            String terminator, String... innerListTerminator) {
        int listLength = objects.size();

        // Special case: Only 2 elements
        if (listLength == 2) {

            // If element is a list, recursive calls
            if (objects.get(0) instanceof List<?>) {
                if (innerListTerminator.length < 1) {
                    appendCommaList((List<?>) objects.get(0), builder, "and");
                    builder.append(" " + terminator + " ");
                    appendCommaList((List<?>) objects.get(1), builder, "and");
                } else {
                    appendCommaList((List<?>) objects.get(0), builder,
                            innerListTerminator[0], Arrays.copyOfRange(
                                    innerListTerminator, 1,
                                    innerListTerminator.length));
                    builder.append(" " + terminator + " ");
                    appendCommaList((List<?>) objects.get(1), builder,
                            innerListTerminator[0], Arrays.copyOfRange(
                                    innerListTerminator, 1,
                                    innerListTerminator.length));
                }
            }
```

```
        // Otherwise, just append special case;
        else {
            builder.append(objects.get(0) + " " + terminator + " "
                    + objects.get(1));
        }
        return;
    }

    // Either 1 or fewer elements or more than 2
    else {
        for (int i = 0; i < listLength; i++) {

            // If element is a list, recursive calls
            if (objects.get(i) instanceof List<?>) {
                if (innerListTerminator.length < 1) {
                    appendCommaList((List<?>) objects.get(i), builder,
                            "and");
                } else {
                    appendCommaList((List<?>) objects.get(i), builder,
                            innerListTerminator[0], Arrays.copyOfRange(
                                    innerListTerminator, 1,
                                    innerListTerminator.length));
                }
            }

            // Otherwise, just append
            else {
                builder.append(objects.get(i).toString().trim());
            }

            // Handle comma separators
            if (i < listLength - 2) {
                builder.append(", ");
            } else if (i == listLength - 2) {
                builder.append(", " + terminator + " ");
            }
        }
        return;
    }
}

/**
 * Appends the given list as a String to the given builder separated by the
 * delimiter.
 *
 * @param objects
 *            The list of objects to append to the builder.
 * @param builder
 *            The {@link StringBuilder} to append to.
 * @param delimiter
 *            The delimiter to go between each element of the list.
 */
public static void appendDelimiterList(List<?> objects,
        StringBuilder builder, String delimiter) {
```

```
        Iterator<?> listIterator = objects.listIterator();
        while (listIterator.hasNext()) {
            builder.append(listIterator.next().toString().trim());
            if (listIterator.hasNext()) {
                builder.append(delimiter);
            } else {
                // There is no next, exit the loop to save a check
                break;
            }
        }
    }

    /**
     * Runs the Language Generator.
     *
     * @param args
     *            [0] = If "-t" option it will print out the parse trees of the
     *            entered representations.
     */
    public static void main(String[] args) {
        Generator generator = new Generator();
        boolean showTree = false;
        if (args.length > 0 && args[0].equalsIgnoreCase("-t")) {
            showTree = true;
        }
        try {
            String input = generator.in.readLine();
            while (input != null) {
                try {
                    generator.parser.parse(input);
                    List<Sentence> sentences = generator.parser
                            .generateSentences();
                    for (Sentence sentence : sentences) {
                        System.out.println(sentence);
                    }

                    if (showTree) {
                        System.out.println(generator.parser);
                    }
                } catch (CouldNotParseException e) {
                    System.err.println(e.getMessage());
                }
                input = generator.in.readLine();
            }
        }

        catch (IOException e2) {
            System.err.println("Error: Could not read from STDIN.");
        }

        finally {
            generator.closeStream();
        }
    }
}
```

B.1.2 Parser.java

```
package edu.hawaii.ctfoo.lang_generator;

import java.util.ArrayList;
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.sentence.BuySentence;
import edu.hawaii.ctfoo.lang_generator.sentence.FindGroupSentence;
import edu.hawaii.ctfoo.lang_generator.sentence.FindMoreSentence;
import edu.hawaii.ctfoo.lang_generator.sentence.SellSentence;
import edu.hawaii.ctfoo.lang_generator.sentence.Sentence;

/**
 * Used for parsing semantic representations in String form into
 * {@link ParseToken}s, {@link Tree}s, and ultimately {@link Sentence}s.
 *
 * @author Christopher Foo
 *
 */
public class Parser {
    /**
     * The {@link ParseTokens} resulting from the initial parse of the string
     */
    private List<ParseToken> tokens;

    /**
     * The IDs of parent tokens in the {@link tokens} list.
     */
    private List<String> parentIds;

    /**
     * The parse trees derived from the {@link tokens} list.
     */
    private List<Tree<ParseToken>> parseTrees;

    /**
     * Create a new Parser and initialize all of the lists and fields.
     */
    public Parser() {
        this.tokens = new ArrayList<ParseToken>();
        this.parentIds = new ArrayList<String>();
        this.parseTrees = new ArrayList<Tree<ParseToken>>();
    }

    /**
     * Returns all {@link ParseToken}s from the latest call to
     * {@link parseToTokens} that are direct children (1 level, child not
     * grand-child or further) of the token with the given ID.
     *
     * @param parentId
     *            The ID of the parent token that the direct children should be
     *            found for.
```

```
     * @return A list of {@link ParseToken}s that are the direct children of the
     *         token with the given ID.
     */
    private List<ParseToken> getDirectChildrenTokens(String parentId) {
        ArrayList<ParseToken> children = new ArrayList<ParseToken>();
        for (ParseToken token : this.tokens) {
            if (token.getParent() == parentId
                    || (token.getParent() != null && token.getParent().equals(
                            parentId))) {
                children.add(token);
            }
        }

        return children;
    }

    /**
     * Gets a list of {@link ParseToken}s that are the roots of parsed semantic
     * structures.
     *
     * @return A list of {@link ParseToken}s that are the roots of the parsed
     *         strings.
     */
    public List<ParseToken> getRootTokens() {
        return this.getDirectChildrenTokens(null);
    }

    /**
     * Finds the {@link ParseToken} with the given ID.
     *
     * @param id
     *            The ID of the {@link ParseToken} to find.
     * @return The token with the matching ID or null if it was not found.
     */
    public ParseToken findId(String id) {
        if (id == null) {
            return null;
        }
        for (ParseToken token : this.tokens) {
            if (token.getId() != null && token.getId().equals(id)) {
                return token;
            }
        }
        return null;
    }

    /**
     * Parses the arguments of a single token in a semantic representation
     * passed to the Parser.
     *
     * @param token
     *            The {@link ParseToken} object that is currently being built by
     *            the Parser.
     * @param args
     *            The arguments of the token.
```

```
 * @throws CouldNotParseException
 *             If the arguments do not match the expected patterns.
 */
private void parseArgs(ParseToken token, String[] args)
        throws CouldNotParseException {
    if (args.length > 2) {
        throw new CouldNotParseException();
    } else if (args.length == 1) {
        String arg = args[0].trim();
        if (Character.isLowerCase(arg.charAt(0))
                && !this.parentIds.contains(arg)) {
            token.setId(arg);
            this.parentIds.add(arg);
        }

        else {
            throw new CouldNotParseException();
        }
    }

    else if (args.length == 2) {
        String arg1 = args[0].trim();
        String arg2 = args[1].trim();
        // Must have a parent in this case, the first argument is the ID of
        // the parent
        if (Character.isLowerCase(arg1.charAt(0))
                && this.parentIds.contains(arg1.trim())) {
            token.setParent(arg1);

            // If it is a parent itself, the second argument will be a lower
            // case
            if (Character.isLowerCase(arg2.charAt(0))) {

                // Can't have duplicates with the same ID
                if (this.parentIds.contains(arg2)) {
                    throw new CouldNotParseException();
                }

                else {
                    token.setId(arg2);
                    this.parentIds.add(arg2);
                }
            }

            else {
                token.setValue(arg2.replace("\"", "").replace("'", "")
                        .trim());
            }
        }

        else {
            throw new CouldNotParseException();
        }
    }
```

```
    else {
        throw new CouldNotParseException();
    }
}

/**
 * Parses the given semantic representation into tokens stored in the
 * {@link tokens} field.
 *
 * @param string
 *            The semantic representation of the sentence.
 * @throws CouldNotParseException
 *             The given string does not match the expected semantic
 *             representation.
 */
private void parseToTokens(String string) throws CouldNotParseException {

    // Make sure lists are clear
    this.parentIds.clear();
    this.tokens.clear();

    String stringCopy = new String(string).trim();
    int foundIndex;
    String[] args;
    while (!stringCopy.isEmpty()) {
        ParseToken token = new ParseToken();
        switch (stringCopy.charAt(0)) {
        case ',':
            token.setLogic(LogicOp.AND);

            // Check for negation after the AND
            int i = 1;

            // Skip the whitespace
            char currentChar = stringCopy.charAt(i);
            while (currentChar == ' ' || currentChar == '\t') {
                currentChar = stringCopy.charAt(++i);
            }
            if (currentChar == '-') {
                token.setNegated(true);
                stringCopy = stringCopy.substring(i + 1).trim();
            } else {
                stringCopy = stringCopy.substring(1).trim();
            }
            break;
        case ';':
            token.setLogic(LogicOp.OR);

            // Check for negation after the OR
            i = 1;

            // Skip the whitespace
            currentChar = stringCopy.charAt(i);
            while (currentChar == ' ' || currentChar == '\t') {
                currentChar = stringCopy.charAt(++i);
```

```
            }
            if (currentChar == '-') {
                token.setNegated(true);
                stringCopy = stringCopy.substring(i + 1).trim();
            } else {
                stringCopy = stringCopy.substring(1).trim();
            }
            break;
        case '-':
            token.setNegated(true);
            stringCopy = stringCopy.substring(1).trim();
            break;
        }

        foundIndex = stringCopy.indexOf('(');

        // No ( for the start of the argument list, syntax error
        if (foundIndex == -1) {
            throw new CouldNotParseException("Error: Syntax error in \""
                    + stringCopy + "\".  Could not parse.");
        }

        token.setType(Character.toUpperCase(stringCopy.charAt(0))
                + stringCopy.substring(1, foundIndex));
        stringCopy = stringCopy.substring(foundIndex + 1).trim();

        foundIndex = stringCopy.indexOf(')');

        // No ) for the end of the argument list, syntax error
        if (foundIndex == -1) {
            throw new CouldNotParseException("Error: Syntax error in \""
                    + stringCopy + "\".  Could not parse.");
        }

        // Parse arguments
        args = stringCopy.substring(0, foundIndex).split(",");
        try {
            parseArgs(token, args);
        }

        catch (CouldNotParseException e) {
            throw new CouldNotParseException("Error: Syntax error in \""
                    + stringCopy + "\".  Could not parse.");
        }

        this.tokens.add(token);
        stringCopy = stringCopy.substring(foundIndex + 1).trim();
    }

    // Empty parentIds list, do not need it until parseToTokens is called
    // again.
    this.parentIds.clear();
}

/**
```

```java
 * Recursive function used to parse the {@link tokens} into {@link Tree}
 * structures in the {@link parseTrees} field.
 *
 * @param parent
 *            The parent of current token.
 * @param current
 *            The token that to parse into a Tree.
 */
private void parseToTree(Tree<ParseToken> parent, ParseToken current) {
    Tree<ParseToken> thisNode;
    if (parent == null) {
        thisNode = new Tree<ParseToken>(current, null);
        this.parseTrees.add(thisNode);
    } else {
        thisNode = parent.addChild(current);
    }

    String rootId = current.getId();
    if (rootId != null) {
        for (ParseToken child : this.getDirectChildrenTokens(rootId)) {
            this.parseToTree(thisNode, child);
        }
    }
}

/**
 * Parses the entire {@link tokens} list into {@link Tree} structures stored
 * in the {@link parseTree} field.
 */
private void parseToTree() {

    // Make sure the parseTree is clear
    this.parseTrees.clear();
    for (ParseToken root : this.getRootTokens()) {
        parseToTree(null, root);
    }

    // The tokens list is not needed any more
    this.tokens.clear();
}

/**
 * Parses the given semantic representation into its {@link Tree} form.
 *
 * @param string
 *            The semantic representation in string form.
 * @throws CouldNotParseException
 *             If the given string does not match the expected semantic
 *             representation.
 */
public void parse(String string) throws CouldNotParseException {
    this.parseToTokens(string); // Remove any whitespace
    this.parseToTree();
}
```

```java
/**
 * Generates all of the {@link Sentence}s for the most recent parse.
 *
 * @return A list of Sentences for the most recent parse.
 */
public List<Sentence> generateSentences() {
    ArrayList<Sentence> sentences = new ArrayList<Sentence>();
    for (Tree<ParseToken> root : this.parseTrees) {
        String type = root.getNode().getType();
        if (type.equalsIgnoreCase("buy")) {
            sentences.add(new BuySentence(root));
        }

        else if (type.equalsIgnoreCase("sell")) {
            sentences.add(new SellSentence(root));
        }

        else if (type.equalsIgnoreCase("findgroup")) {
            sentences.add(new FindGroupSentence(root));
        }

        else if (type.equalsIgnoreCase("findmore")) {
            sentences.add(new FindMoreSentence(root));
        }

        else {
            // Unrecognized sentence, ignore.
        }
    }
    return sentences;
}

/**
 * Recursive function that prints the structure of the parse based on the
 * tokens parsed.
 *
 * @param indentation
 *            The current indentation level for this token.
 * @param currentToken
 *            The current token.
 * @return A string representing the structure of the this token and its
 *         children from the parse.
 */
private String printParse(String indentation, ParseToken currentToken) {
    StringBuilder returnString = new StringBuilder();

    returnString.append(indentation + currentToken.toString() + "\n");
    if (currentToken.getId() != null) {
        for (ParseToken token : this.getDirectChildrenTokens(currentToken
                .getId())) {

            returnString.append(printParse(indentation + "\t", token));
        }
    }
```

```
        return returnString.toString();
    }

    @Override
    /**
     * If the tree is parsed, it will print the semantic tree based on the parse tree.
     * If the tree is not parsed, but the tokens are it will print the semantic tree
based
     * on the tokens.
     */
    public String toString() {
        StringBuilder returnString = new StringBuilder();
        if (this.parseTrees.size() > 0) {
            for (Tree<ParseToken> tree : this.parseTrees) {
                returnString.append(tree.toString() + "\n");
            }
        }

        else {

            for (ParseToken root : this.getRootTokens()) {
                returnString.append(this.printParse("", root));
            }
        }
        return returnString.toString();
    }
}
```

B.1.3 CouldNotParseException.java

```
package edu.hawaii.ctfoo.lang_generator;

/**
 * An {@link Exception} to be thrown when the parser cannot parse the given
 * string.
 *
 * @author Christopher Foo
 *
 */
public class CouldNotParseException extends Exception {

    /**
     * Automatically generated ID.
     */
    private static final long serialVersionUID = 8075760826309446797L;

    /**
     * Creates a new blank CouldNotParseException.
     */
    public CouldNotParseException() {
        super();
    }

    /**
```

```
     * Creates a new CouldNotParseException with the given message.
     *
     * @param message
     *            The message for the Exception.
     */
    public CouldNotParseException(String message) {
        super(message);
    }

}
```

B.1.4 ParseToken.java

```
package edu.hawaii.ctfoo.lang_generator;
/**
 * An object representing a single token from the parsed semantic representation.
 * @author Christopher Foo
 *
 */
public class ParseToken {
    /**
     * The type of the token (i.e. Subject, Object, Item, etc)
     */
    private String type = null;

    /**
     * The ID of the token's parent.  null if the token does not have a parent.
     */
    private String parent = null;

    /**
     * The ID of the token which is used to associate it with its children.
     * null if the token does not have any children.
     */
    private String id = null;

    /**
     * The value of the token.  null if the token is a parent token.
     */
    private String value = null;

    /**
     * The logic operation ({@link LogicOp}) used to connect this token with the
     * others.
     */
    private LogicOp logic = null;

    /**
     * If the token was negated or not.
     */
    private boolean negated = false;

    /**
     * Gets the type of the token (i.e. Subject, Object, Item, etc).
```

```
     * @return The type of the token.
     */
    public String getType() {
        return this.type;
    }
    /**
     * Sets the type of the token (i.e. Subject, Object, Item, etc) to the given
     * value.
     * @param type The new type of the token.
     */
    public void setType(String type) {
        this.type = type;
    }
    /**
     * Gets the ID of the token's parent. null if the token does not have a parent.
     * @return The ID of the token's parent.  null if the token does not have a
*                parent.
     */
    public String getParent() {
        return this.parent;
    }
    /**
     * Sets the ID of the token's parent to the given value.
     * @param parent The ID of the token's new parent.  null indicates no parent.
     */
    public void setParent(String parent) {
        this.parent = parent;
    }
    /**
     * Gets the ID of the token. null if the token is not a parent.
     * @return The ID of the token.  null if the token is not a parent.
     */
    public String getId() {
        return this.id;
    }
    /**
     * Sets the ID of the token to the given value.
     * @param id The new ID of the token.  null indicates that it is not a parent.
     */
    public void setId(String id) {
        this.id = id;
    }
    /**
     * Gets the value of the token.
     * @return The value of the token.  null if the token is a parent.
     */
    public String getValue() {
        return this.value;
    }
    /**
     * Sets the value of the token to the given value.
     * @param value The new value for the token.
     */
    public void setValue(String value) {
        this.value = value;
```

```
}
/**
 * Gets the logic operation used to link this token with the others.
 * @return The logic operation.
 */
public LogicOp getLogic() {
    return this.logic;
}
/**
 * Sets the logic operation used to link this token with the others.
 * @param logic The new logic operation of the token.
 */
public void setLogic(LogicOp logic) {
    this.logic = logic;
}

/**
 * Gets if the token was negated or not.
 * @return If the token was negated.
 */
public boolean isNegated() {
    return negated;
}
/**
 * Sets whether the token was negated or not.
 * @param negated If the token was negated (true = negated).
 */
public void setNegated(boolean negated) {
    this.negated = negated;
}
@Override
/**
 * Returns a String of the ParseToken in JSON form.
 */
public String toString() {
    StringBuilder returnString = new StringBuilder();
    String fieldValue;
    fieldValue = (this.type == null) ? "null, " : this.type + ", ";
    returnString.append("{ Type: " + fieldValue);

    fieldValue = (this.id == null) ? "null, " : this.id + ", ";
    returnString.append("ID: " + fieldValue);

    fieldValue = (this.parent == null) ? "null, " : this.parent + ", ";
    returnString.append("Parent: " + fieldValue);

    fieldValue = (this.value == null) ? "null, " : this.value + ", ";
    returnString.append("Value: " + fieldValue);

    fieldValue = (this.logic == null) ? "null, " : this.logic + ", ";
    returnString.append("Logic Operation: " + fieldValue);

    fieldValue = (this.negated) ? "true }" : "false }";
    returnString.append("Negated: " + fieldValue);
```

```
        return returnString.toString();
    }
}
```

B.1.5 LogicOp.java

```
package edu.hawaii.ctfoo.lang_generator;

/**
 * Represents the two logical operations (AND & OR) in the semantic
 * representations of the Language Generator.
 *
 * @author Christopher Foo
 *
 */
public enum LogicOp {
    AND, OR;

    @Override
    public String toString() {
        switch (this) {
        case AND:
            return "AND";
        case OR:
            return "OR";
        default:
            return "Undefined LogicOp";
        }
    }
}
```

B.1.6 Tree.java

```
package edu.hawaii.ctfoo.lang_generator;

import java.util.ArrayList;
import java.util.List;

/**
 * A node in a tree structure. An entire tree structure is made out of these
 * nodes connected by their parent / children relationships.
 *
 * @author Christopher Foo
 *
 * @param <T>
 *            The type of the nodes in the Tree.
 */
public class Tree<T> {

    /**
     * The node at the top of this tree.
     */
    private T node;
```

```java
/**
 * This node's parent.
 */
private Tree<T> parent;

/**
 * A list of children nodes.
 */
private List<Tree<T>> children;

/**
 * Creates a new Tree with the given value for its node and given parent.
 *
 * @param node
 *            The value of the node.
 * @param parent
 *            The node's parent. null if it has no parent.
 */
public Tree(T node, Tree<T> parent) {
    this.node = node;
    this.parent = parent;
    this.children = new ArrayList<Tree<T>>();
}

/**
 * Gets the value of the node.
 *
 * @return The value of the node.
 */
public T getNode() {
    return node;
}

/**
 * Sets the value of the node to the given value.
 *
 * @param node
 *            The new value of the node.
 */
public void setNode(T node) {
    this.node = node;
}

/**
 * Gets the parent of the node.
 *
 * @return The node's parent. null if the node does not have a parent.
 */
public Tree<T> getParent() {
    return parent;
}

/**
 * Sets the parent of the node.
 *
```

```java
     * @param parent
     *            The parent of the node. null indicates that it has no parent.
     */
    public void setParent(Tree<T> parent) {
        this.parent = parent;
    }

    /**
     * Gets the direct children of the node (1 level).
     *
     * @return The direct children of this node.
     */
    public List<Tree<T>> getDirectChildren() {
        return children;
    }

    /**
     * Adds a child to this node.
     *
     * @param child
     *            The child node to add to this node.
     * @return The added node.
     */
    public Tree<T> addChild(Tree<T> child) {
        this.children.add(child);
        child.parent = this;
        return child;
    }

    /**
     * Adds a child to this node.
     *
     * @param childValue
     *            The value of the child node to add to this node.
     * @return The added node.
     */
    public Tree<T> addChild(T childValue) {
        Tree<T> child = new Tree<T>(childValue, this);
        this.children.add(child);
        return child;
    }

    /**
     * Finds the direct children of this node that match the matcher with the
     * given key.
     *
     * @param key
     *            The key to match with the matcher.
     * @param matcher
     *            The {@link MatchFunctor} used to match the key with a node.
     * @return A list of matching children nodes.
     */
    public <U> List<Tree<T>> findDirect(U key, MatchFunctor<U, T> matcher) {
        ArrayList<Tree<T>> matches = new ArrayList<Tree<T>>();
        for (Tree<T> child : this.children) {
```

```
        if (matcher.match(key, child.node)) {
            matches.add(child);
        }
    }

    return matches;
}

/**
 * Finds all children (and their children, etc.) that match the matcher with
 * the given key.
 *
 * @param key
 *            The key to match with the matcher.
 * @param matcher
 *            The {@link MatchFunctor} used to match the key with a node.
 * @return A list of all matching nodes below this node.
 */
public <U> List<Tree<T>> findAll(U key, MatchFunctor<U, T> matcher) {
    ArrayList<Tree<T>> matches = new ArrayList<Tree<T>>();
    for (Tree<T> child : this.children) {
        if (matcher.match(key, child.node)) {
            matches.add(child);
        }
        matches.addAll(child.findAll(key, matcher));
    }
    return matches;
}

/**
 * Recursively builds the string to represent this {@link Tree} and its
 * children (and thier children, etc.)
 *
 * @param indent
 *            The current level of indentation for this level of nodes.
 * @return A string of the Tree's structure.
 */
private String toStringHelper(String indent) {
    StringBuilder builder = new StringBuilder();
    builder.append(indent + this.node.toString() + "\n");

    for (Tree<T> child : this.children) {
        builder.append(child.toStringHelper(indent + "\t"));
    }

    return builder.toString();
}

@Override
/**
 * Returns a String representing this node and all of its children (and their
 * children, etc)
 */
public String toString() {
    return this.toStringHelper("");
```

```
    }
}
```

B.1.7 MatchFunctor.java

```
package edu.hawaii.ctfoo.lang_generator;

/**
 * A functor used to find matching nodes in a {@link Tree}.
 *
 * @author Christopher Foo
 *
 * @param <T>
 *            The type of the key to match with.
 * @param <U>
 *            The type of the nodes that will be searched.
 */
public interface MatchFunctor<T, U> {

    /**
     * Matches the given node with the given key.
     *
     * @param key
     *            The key to be matched.
     * @param node
     *            The node to be checked.
     * @return If the key and the node match or not.
     */
    public boolean match(T key, U node);
}
```

B.1.8 TypeMatcher.java

```
package edu.hawaii.ctfoo.lang_generator;

/**
 * An implementation of the {@link MatchFunctor} used for finding ParseTokens
 * with the given String keys.
 *
 * @author Christopher Foo
 *
 */
public class TypeMatcher implements MatchFunctor<String, ParseToken> {

    @Override
    /**
     * Matches if the given searchItem's type and the key match.
     */
    public boolean match(String key, ParseToken searchItem) {
        if (searchItem == null) {
            return false;
        }

        if (searchItem.getType() == key
```

```
                || searchItem.getType().equalsIgnoreCase(key)) {
            return true;
        }

        else {
            return false;
        }
    }

}
```

B.2 edu.hawaii.ctfoo.lang_generator.entity

B.2.1 Entity.java

```
package edu.hawaii.ctfoo.lang_generator.entity;

import edu.hawaii.ctfoo.lang_generator.LogicOp;

/**
 * An entity in a sentence.
 *
 * @author Christopher Foo
 *
 */
public abstract class Entity {

    /**
     * The logic operation connecting this entity to the others.
     */
    protected LogicOp logic;

    /**
     * If the entity was negated.
     */
    protected boolean negated;

    /**
     * Creates a new Entity with default values.
     */
    public Entity() {
        this.logic = null;
        this.negated = false;
    }

    /**
     * Gets the logic operation used to connect this entity with the others.
     *
     * @return The logic operation.
     */
    public LogicOp getLogic() {
        return this.logic;
    }
```

```
    /**
     * Checks if this entity was negated.
     *
     * @return If the entity was negated.
     */
    public boolean isNegated() {
        return this.negated;
    }
}
```

B.2.2 Instance.java

```
package edu.hawaii.ctfoo.lang_generator.entity;

import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;

/**
 * An entity representing an Instance (i.e. dungeon) in a game.
 * @author Christopher Foo
 *
 */
public class Instance extends Entity {

    /**
     * The name of the instance.
     */
    private String name;

    /**
     * The mode of the instance.
     */
    private String mode;

    /**
     * The difficulty level of the instance.
     */
    private String difficulty;

    /**
     * Creates a new Instance with default values.
     */
    public Instance() {
        super();
        this.name = "";
        this.mode = "";
        this.difficulty = "";
    }

    /**
     * Creates a new Instance based on the given {@link Tree}.
     * @param instanceToken The Tree of an Instance token node.
     */
    public Instance(Tree<ParseToken> instanceToken) {
```

```
    this();
    if(instanceToken.getNode().getType().equalsIgnoreCase("instance")) {
       if(instanceToken.getNode().isNegated()) {
            this.negated = true;
        }

       String type;
       String value;

       // Read all of the attributes for the instance into the appropriate
       // fields
       for (Tree<ParseToken> instanceAttribute :
               InstanceToken.getDirectChildren()) {

           type = instanceAttribute.getNode().getType();

           value = (instanceAttribute.getNode().getValue() == null) ? null
                   : instanceAttribute.getNode().getValue();

           // Set name
           if(type.equalsIgnoreCase("name")) {
               this.name = value;
           }

           // Set mode
           else if(type.equalsIgnoreCase("mode")) {
               this.mode = value;
           }

           // Set difficulty
           else if(type.equalsIgnoreCase("difficulty")) {
               this.difficulty = value;
           }

           // Ignore unrecognized attributes
       }
    }
}

/**
 * Gets the name of the Instance.
 * @return The name of the Instance.
 */
public String getName() {
    return this.name;
}

/**
 * Gets the mode of the Instance.
 * @return The mode of the Instance.
 */
public String getMode() {
    return this.mode;
}
```

```
    /**
     * Gets the difficulty level of the Instance.
     * @return The difficulty level of the Instance.
     */
    public String getDifficulty() {
        return this.difficulty;
    }

    @Override
    /**
     * Returns a String representation of this Instance.
     */
    public String toString() {
        if (this.name.equals("")) {
            return "< Error Incomplete Instance Encountered >";
        }
        StringBuilder builder = new StringBuilder();
        if (!this.difficulty.equals("")) {
            builder.append(this.difficulty + " ");
        }

        if (!this.mode.equals("")) {
            builder.append(this.mode + " ");
        }

        builder.append(this.name);
        return builder.toString().trim();
    }
}
```

B.2.3 Item.java

```
package edu.hawaii.ctfoo.lang_generator.entity;

import java.util.ArrayList;
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.MatchFunctor;
import edu.hawaii.ctfoo.lang_generator.Generator;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.TypeMatcher;

public class Item extends Entity {
    /**
     * The name of the item.
     */
    private String name;

    /**
     * The rarity / quality of the item.
     */
    private String rarity;
```

```java
/**
 * The type of the item (i.e. sword, shield, helmet, etc).
 */
private String type;

/**
 * The value of the item as a list of {@link MoneyAmount} objects.
 */
private List<MoneyAmount> value;

/**
 * The level of the item.
 */
private int level;

/**
 * The quantity of this item to buy / sell.
 */
private int quantity;

/**
 * Creates a new Item with default values.
 */
public Item() {
    super();
    this.name = "";
    this.rarity = "";
    this.type = "";
    this.value = new ArrayList<MoneyAmount>();
    this.level = -1;
    this.quantity = 1;
}

/**
 * Creates a new Item object based on the given Item {@link ParseToken} in
 * the parse tree. Returns a default Item object if the given node is not an
 * Item node.
 *
 * @param itemToken
 *            The item token to parse from.
 */
public Item(Tree<ParseToken> itemToken) {

    // Set everything to defaults
    this();

    // If item token, read in attributes
    if (itemToken.getNode().getType().equalsIgnoreCase("item")) {

        if (itemToken.getNode().isNegated()) {
            this.negated = true;
        }

        String type;
        String value;
```

```
// Read all of the attributes for the item into the appropriate
// fields
for (Tree<ParseToken> itemAttribute : itemToken.getDirectChildren()) {

    type = itemAttribute.getNode().getType();

    value = (itemAttribute.getNode().getValue() == null) ? null
            : itemAttribute.getNode().getValue();

    // Set Name
    if (type.equalsIgnoreCase("name")) {
        this.name = value;
    }

    // Set Rarity
    else if (type.equalsIgnoreCase("rarity")) {
        this.rarity = value;
    }

    // Set Type
    else if (type.equalsIgnoreCase("type")) {
        this.type = value;
    }

    // Set Level
    else if (type.equalsIgnoreCase("level")) {
        try {
            this.level = Integer.parseInt(value);
        } catch (NumberFormatException e) {
            System.err.println("Error: Could not parse \"" + value
                    + "\" as an integer.");
        }
    }

    // Set Quantity
    else if (type.equalsIgnoreCase("quantity")) {
        try {
            this.quantity = Integer.parseInt(value);
        } catch (NumberFormatException e) {
            System.err.println("Error: Could not parse " + value
                    + " as an integer.");
        }
    }

    // Set Value
    else if (type.equalsIgnoreCase("value")) {
        MatchFunctor<String, ParseToken> typeMatcher = new TypeMatcher();

        // Get all underlying MoneyAmount elements
        for (Tree<ParseToken> moneyAmount : itemAttribute.findAll(
                "moneyamount", typeMatcher)) {

            MoneyAmount amount = new MoneyAmount(moneyAmount);
```

```
                    // If the MoneyAmount is complete add it, if not ignore
                    // it
                    if (amount.getDenomination() != -1
                            && !amount.getCurrency().equals("")) {
                        this.addValue(amount);
                    }
                }
            }

            // Ignore unrecognized attributes
        }
    }
}

/**
 * Gets the name of the Item.
 *
 * @return The name of the Item.
 */
public String getName() {
    return name;
}

/**
 * Gets the rarity / quality of the Item.
 *
 * @return The rarity / quality of the Item.
 */
public String getRarity() {
    return rarity;
}

/**
 * Gets the type of the Item (i.e. sword, shield, helmet, boots, etc.)
 *
 * @return The type of the item.
 */
public String getType() {
    return type;
}

/**
 * Gets the value of the Item as an array of {@link MoneyAmount} objects.
 *
 * @return The value of the Item.
 */
public MoneyAmount[] getValue() {
    return (MoneyAmount[]) this.value.toArray();
}

/**
 * Adds the given amount to the Item's value.
 *
 * @param denomination
 *            The denomination of the added amount.
```

```
     * @param currency
     *            The currency of the added amount.
     */
    public void addValue(double denomination, String currency) {
        this.value.add(new MoneyAmount(denomination, currency));
    }

    /**
     * Adds the given amount to the Item's value.
     *
     * @param amount
     *            The {@link MoneyAmount} to add.
     */
    public void addValue(MoneyAmount amount) {
        this.value.add(amount);
    }

    /**
     * Gets the level of the Item.
     *
     * @return The level of the Item.
     */
    public int getLevel() {
        return level;
    }

    /**
     * Gets the quantity of the Item.
     *
     * @return The quantity of the Item.
     */
    public int getQuantity() {
        return quantity;
    }

    @Override
    /**
     * Returns a String representation of the Item.
     */
    public String toString() {
        StringBuilder returnString = new StringBuilder();

        // Append quantity if it is there.
        if (this.quantity > 0) {
            returnString.append(this.quantity + " ");
        }

        // If has name, use it.
        if (!this.name.equals("")) {

            if (this.quantity > 1) {
                returnString.append(Generator.pluralize(this.name));
            } else {
                returnString.append(this.name);
            }
```

```
            returnString.append(" ");

            if (this.value.size() > 0) {
                returnString.append("for ");
            }

            Generator.appendCommaList(this.value, returnString, "and");
            return returnString.toString().trim();

        }

        // Otherwise use type if it has it.
        else if (!this.type.equals("")) {
            if (!this.rarity.equals("")) {
                returnString.append(this.rarity.toLowerCase() + " ");
            }
            if (this.level > -1) {
                returnString.append("item level " + this.level + " ");
            }
            if (this.quantity > 1) {
                returnString
                        .append(Generator.pluralize(this.type.toLowerCase()));
            } else {
                returnString.append(this.type.toLowerCase());

            }
            returnString.append(" ");

            if (this.value.size() > 0) {
                returnString.append("for ");
            }

            Generator.appendCommaList(this.value, returnString, "and");
            return returnString.toString().trim();
        }

        // Item needs a name or type to be complete.
        else {
            return "< Error: Incomplete item encountered >";
        }
    }
}
```

B.2.4 MoneyAmount.java

```
package edu.hawaii.ctfoo.lang_generator.entity;

import edu.hawaii.ctfoo.lang_generator.MatchFunctor;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.TypeMatcher;

/**
```

```java
 * An amount of money consisting of a denomination and its currency.
 *
 * @author Christopher Foo
 *
 */
public class MoneyAmount {

    /**
     * The denomination of the amount.
     */
    private double denomination = -1;

    /**
     * The currency of the money amount.
     */
    private String currency = "";

    /**
     * Creates a new MoneyAmount with the given denomination and currency.
     *
     * @param denomination
     *            The denomination of the amount.
     * @param currency
     *            The currency of the amount.
     */
    public MoneyAmount(double denomination, String currency) {
        this.denomination = denomination;
        this.currency = currency;
    }

    /**
     * Creates a new MoneyAmount token by parsing the {@link Tree} from the
     * given money token.
     *
     * @param moneyToken
     *            The node containing the money amount token.
     */
    public MoneyAmount(Tree<ParseToken> moneyToken) {

        if (moneyToken.getNode().getType().equalsIgnoreCase("moneyamount")) {
            MatchFunctor<String, ParseToken> typeMatcher = new TypeMatcher();

            // Get the denomination for the MoneyAmount (defaults to
            // last one if there are multiple)
            for (Tree<ParseToken> denominationToken : moneyToken.findAll(
                    "denomination", typeMatcher)) {
                try {
                    this.denomination = Integer.parseInt(denominationToken
                            .getNode().getValue());
                } catch (NumberFormatException e) {
                    System.err.println("Error: Could not parse "
                            + denominationToken.getNode().getValue()
                            + " as an integer.");
                }
            }
```

```
            // Get the currency for the MoneyAmount (defaults to the
            // last one if there are multiple)
            for (Tree<ParseToken> currencyToken : moneyToken.findAll(
                    "currency", typeMatcher)) {
                this.currency = currencyToken.getNode().getValue();
            }
        }
    }

    /**
     * Gets the denomination of the amount.
     *
     * @return The denomination of the amount.
     */
    public double getDenomination() {
        return denomination;
    }

    /**
     * Gets the currency of the amount.
     *
     * @return The currency of the amount.
     */
    public String getCurrency() {
        return currency;
    }

    @Override
    /**
     * Returns a String representation of the MoneyAmount.
     */
    public String toString() {
        if (this.denomination - (-1.0) < 0.0001 || this.currency.equals("")) {
            return "< Error: Incomplete currency encountered >";
        } else {

            if (this.denomination > 1000) {
                return (this.denomination / 1000) + "k "
                        + this.currency.toLowerCase();
            } else {
                return this.denomination + " " + this.currency.toLowerCase();
            }

        }
    }

}
```

B.2.5 Player.java

```
package edu.hawaii.ctfoo.lang_generator.entity;

import java.util.ArrayList;
```

```java
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.Generator;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;

/**
 * An Entity representing a player character in a MMORPG.
 *
 * @author Christopher Foo
 *
 */
public class Player extends Entity {

    /**
     * The classes of the player character.
     */
    private List<String> characterClasses;

    /**
     * The specializations of the player character's classes.
     */
    private List<String> characterSpecializations;

    /**
     * The race of the player character.
     */
    private String characterRace;

    /**
     * The role of the player character.
     */
    private String role;

    /**
     * The level of the player character.
     */
    private int level;

    /**
     * The average item level of the player character's items.
     */
    private int itemLevel;

    /**
     * The number of player characters.
     */
    private int quantity;

    /**
     * Creates a new Player with the default values.
     */
    public Player() {
        super();
        this.characterClasses = new ArrayList<String>();
```

```
        this.characterSpecializations = new ArrayList<String>();
        this.characterRace = "";
        this.role = "";
        this.level = -1;
        this.itemLevel = -1;
        this.quantity = 1;
    }

    /**
     * Creates a new Player based on the information in the {@link Tree} rooted
     * by the given player token.
     *
     * @param playerToken
     *            The player token in the parse tree to build the Player from.
     */
    public Player(Tree<ParseToken> playerToken) {
        this();

        if (playerToken.getNode().getType().equalsIgnoreCase("player")) {
            if (playerToken.getNode().isNegated()) {
                this.negated = true;
            }

            String type;
            String value;

            // Read all of the attributes for the player into the appropriate
            // fields
            for (Tree<ParseToken> playerAttributes : playerToken
                    .getDirectChildren()) {

                type = playerAttributes.getNode().getType();

                value = (playerAttributes.getNode().getValue() == null) ? null
                        : playerAttributes.getNode().getValue();

                // Set class
                if (type.equalsIgnoreCase("class")) {
                    this.characterClasses.add(value);
                }

                // Set specialization
                else if (type.equalsIgnoreCase("specialization")) {
                    this.characterSpecializations.add(value);
                }

                // Set Race
                else if (type.equalsIgnoreCase("race")) {
                    this.characterRace = value;
                }

                // Set Role
                else if (type.equalsIgnoreCase("role")) {
                    this.role = value;
                }
```

```
            // Set level
            else if (type.equalsIgnoreCase("level")) {
                try {
                    this.level = Integer.parseInt(value);
                } catch (NumberFormatException e) {
                    System.err.println("Error: Could not parse \"" + value
                            + "\" as an integer.");
                }
            } else if (type.equalsIgnoreCase("itemlevel")) {
                try {
                    this.itemLevel = Integer.parseInt(value);
                } catch (NumberFormatException e) {
                    System.err.println("Error: Could not parse \"" + value
                            + "\" as an integer.");
                }
            }

            // Set quantity
            else if (type.equalsIgnoreCase("quantity")) {
                try {
                    this.quantity = Integer.parseInt(value);
                } catch (NumberFormatException e) {
                    System.err.println("Error: Could not parse \"" + value
                            + "\" as an integer.");
                }
            }

            // Ignore other unrecognized attributes
        }
    }
}

/**
 * Gets the classes of the player character.
 *
 * @return The classes of the player character in an array.
 */
public String[] getCharacterClass() {
    return (String[]) this.characterClasses.toArray();
}

/**
 * Adds a new class to the player character.
 *
 * @param characterClass
 *            The new class to add.
 */
public void addCharacterClass(String characterClass) {
    this.characterClasses.add(characterClass);
}

/**
 * Gets the specializations of the player character's classes.
 *
```

```
     * @return The specializations of the player character's classes in an
     *         array.
     */
    public String[] getCharacterSpecialization() {
        return (String[]) this.characterSpecializations.toArray();
    }

    /**
     * Adds a new specialization.
     *
     * @param characterSpecialization
     *            The new specialization ot add.
     */
    public void addCharacterSpecialization(String characterSpecialization) {
        this.characterSpecializations.add(characterSpecialization);
    }

    /**
     * Gets the player character's race.
     *
     * @return the characterRace The player character's race.
     */
    public String getCharacterRace() {
        return this.characterRace;
    }

    /**
     * Gets the player character's role.
     *
     * @return The player character's role.
     */
    public String getRole() {
        return this.role;
    }

    /**
     * Gets the level of the player character.
     *
     * @return The level of the player character.
     */
    public int getLevel() {
        return level;
    }

    /**
     * Gets the average item level of the player character's items.
     *
     * @return The average item level of the player character's items.
     */
    public int getItemLevel() {
        return itemLevel;
    }

    /**
     * Gets the number of this type of player character.
```

```
     *
     * @return The number of this type of player character.
     */
    public int getQuantity() {
        return this.quantity;
    }

    @Override
    /**
     * Returns a String representation of this Player character.
     */
    public String toString() {
        StringBuilder builder = new StringBuilder();
        if (this.characterClasses.size() < 1 && this.role.equals("")) {
            return "< Error: Incomplete player encountered >";
        }

        if (this.quantity > 1) {
            builder.append(this.quantity + " ");
        }
        if (this.level > -1) {
            builder.append("Level " + this.level + " ");
            if (this.itemLevel > -1) {
                builder.append("/ ");
            }
        }

        if (this.itemLevel > -1) {
            builder.append("Item Level " + this.itemLevel + " ");
        }

        if (this.characterClasses.size() > 0) {
            if (!this.characterRace.equals("")) {
                builder.append(this.characterRace + " ");
            }

            if (this.characterSpecializations.size() > 0) {
                Generator.appendDelimiterList(this.characterSpecializations,
                        builder, " / ");
                builder.append(" ");
            }

            if (this.characterClasses.size() == 1 && this.quantity > 1) {
                builder.append(Generator.pluralize(this.characterClasses.get(0)));
            }

            else {
                Generator.appendDelimiterList(this.characterClasses, builder,
                        " / ");
            }
        }

        else {
            if (this.quantity > 1) {
                builder.append(Generator.pluralize(this.role));
```

```
            } else {
                builder.append(this.role);
            }
        }
        return builder.toString().trim();
    }
}
```

B.3 edu.hawaii.ctfoo.lang_generator.sentence

B.3.1 Sentence.java

```
package edu.hawaii.ctfoo.lang_generator.sentence;

import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.LogicOp;
import edu.hawaii.ctfoo.lang_generator.MatchFunctor;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.TypeMatcher;
import edu.hawaii.ctfoo.lang_generator.entity.Entity;

/**
 * Represents a simple English sentence about a single event.
 *
 * @author Christopher Foo
 *
 */
public abstract class Sentence {

    /**
     * The objects (acted upon) of the event. Each sublist is connected by an OR
     * and every element in each sublist is connected by an AND.
     */
    protected List<List<Entity>> eventObject;

    /**
     * The subjects (actors) of the event. Each sublist is connected by an OR
     * and every element in each sublist is connected by an AND.
     */
    protected List<List<Entity>> eventSubject;

    /**
     * An array of valid classes for the Sentence's objects.
     */
    protected Class<?>[] validObjectClasses;

    /**
     * An array of valid classes for the Sentence's subjects.
     */
```

```
protected Class<?>[] validSubjectClasses;

/**
 * Creates and initializes a new Sentence.
 *
 * @param validObjectClasses
 *            An array of the valid classes for the Sentence's objects.
 * @param validSubjectClasses
 *            An array of the valid classes for the Sentence's subjects.
 */
public Sentence(Class<?>[] validObjectClasses,
        Class<?>[] validSubjectClasses) {
    this.validObjectClasses = validObjectClasses;
    this.validSubjectClasses = validSubjectClasses;
    this.eventObject = new ArrayList<List<Entity>>();
    this.eventSubject = new ArrayList<List<Entity>>();
}

/**
 * Adds a new object to the eventObjects list.
 *
 * @param newObject
 *            The new object to add.
 * @param index
 *            The index of the sublist to add it to.
 */
public void addObject(Entity newObject, int index) {
    if (checkObject(newObject)) {
        if (index >= this.eventObject.size()) {
            this.eventObject.add(new ArrayList<Entity>());
        }
        this.eventObject.get(index).add(newObject);
    }
}

/**
 * Adds a new subject to the eventSubjects list.
 *
 * @param newSubject
 *            The new subject to add.
 * @param index
 *            The index of the sublist to add it to.
 */
public void addSubject(Entity newSubject, int index) {
    if (checkSubject(newSubject)) {
        if (index >= this.eventSubject.size()) {
            this.eventSubject.add(new ArrayList<Entity>());
        }
        this.eventSubject.get(index).add(newSubject);
    }
}

/**
 * Reads in all of the subjects or objects for the given sentence.
 *
```

```
     * @param sentenceToken
     *            The node containing the token for the sentence.
     * @param type
     *            Either "subject" to read in the subjects or "object" to read
     *            in the objects.
     */
    public void readSubObj(Tree<ParseToken> sentenceToken, String type) {
        MatchFunctor<String, ParseToken> typeMatcher = new TypeMatcher();

        if (type.equalsIgnoreCase("subject") || type.equalsIgnoreCase("object")) {

            List<List<Entity>> targetList;
            if (type.equalsIgnoreCase("subject")) {
                targetList = this.eventSubject;
            } else {
                targetList = this.eventObject;
            }

            int index = 0;
            for (Tree<ParseToken> typeNode : sentenceToken.findAll(type,
                    typeMatcher)) {
                ParseToken typeToken = typeNode.getNode();

                // Put in new list if OR
                if (typeToken.getLogic() == LogicOp.OR
                        && (index >= targetList.size() || targetList.get(index)
                                .size() > 0)) {
                    index++;
                }

                // Get all of the subject or object entities
                for (Tree<ParseToken> typeChild : typeNode.getDirectChildren()) {
                    ParseToken tokenChild = typeChild.getNode();
                    Entity entity = null;
                    if (typeToken.isNegated()) {
                        tokenChild.setNegated(true);
                    }

                    // Use reflection to get a new instance of the entity
                    try {
                        Class<? extends Entity> foundClass = Class.forName(
                                "edu.hawaii.ctfoo.lang_generator.entity."
                                        + typeChild.getNode().getType())
                                .asSubclass(Entity.class);
                        Constructor<? extends Entity> foundClassConstructor
                                = foundClass.getConstructor(Tree.class);
                        entity = foundClassConstructor.newInstance(typeChild);
                        if (entity != null) {

                            // Update index if OR
                            if (tokenChild.getLogic() == LogicOp.OR
                                    && (index >= targetList.size() || targetList
                                            .get(index).size() > 0)) {
                                index++;
                            }
```

```
                        // Add to the appropriate list
                        if (type.equalsIgnoreCase("object")) {
                            this.addObject(entity, index);
                        } else {
                            this.addSubject(entity, index);
                        }
                    }
                } catch (ClassNotFoundException e) {
                    // Ignore it
                } catch (IllegalArgumentException e) {
                    // Ignore it
                } catch (InstantiationException e) {
                    // Ignore it
                } catch (IllegalAccessException e) {
                    // Ignore it
                } catch (InvocationTargetException e) {
                    // Ignore it
                } catch (SecurityException e) {
                    // Ignore it
                } catch (NoSuchMethodException e) {
                    // Ignore it
                }
            }
        }
    }
}

/**
 * Checks the objects of the Sentence to ensure that they are all of valid
 * classes.
 *
 * @return If all of the objects are valid. Returns false if at least one is
 *         invalid.
 */
protected boolean checkObject() {
    for (List<Entity> objectList : this.eventObject) {
        for (Entity objectElement : objectList) {
            if (!checkObject(objectElement)) {
                return false;
            }
        }
    }
    return true;
}

/**
 * Checks if the given {@link Entity} is a valid class for the Sentence's
 * objects.
 *
 * @param object
 *            The Entity to check.
 * @return If the Entity is a valid object class.
 */
protected boolean checkObject(Entity object) {
```

```
        for (Class<?> validClass : this.validObjectClasses) {
            if (validClass.isInstance(object)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Checks all of the subjects of the Sentence to ensure that they are all of
     * valid classes for the Sentence's subjects.
     *
     * @return If all of the subjects are valid. Returns false if at least one
     *         of the subjects is of an invalid class.
     */
    protected boolean checkSubject() {
        for (List<Entity> subjectList : this.eventSubject) {
            for (Entity subjectElement : subjectList) {
                if (!checkSubject(subjectElement)) {
                    return false;
                }
            }
        }
        return true;
    }

    /**
     * Checks if the given {@link Entity} is a valid class for this Sentence's
     * subjects.
     *
     * @param subject
     *            The Entity to check.
     * @return If the given Entity is a valid subject class.
     */
    protected boolean checkSubject(Entity subject) {
        for (Class<?> validClass : this.validSubjectClasses) {
            if (validClass.isInstance(subject)) {
                return true;
            }
        }
        return false;
    }
}
```

B.3.2 BuySentence.java

```
package edu.hawaii.ctfoo.lang_generator.sentence;

import java.util.ArrayList;
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.MatchFunctor;
import edu.hawaii.ctfoo.lang_generator.Generator;
import edu.hawaii.ctfoo.lang_generator.LogicOp;
```

```java
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.TypeMatcher;
import edu.hawaii.ctfoo.lang_generator.entity.Item;
import edu.hawaii.ctfoo.lang_generator.entity.Player;

/**
 * A {@link Sentence} where the event is buying {@link Item}s.
 * @author Christopher Foo
 *
 */
public class BuySentence extends Sentence {

    /**
     * The array of valid classes for the BuySentence's objects.
     */
    private static final Class<?>[] validObjectClasses = { Item.class };

    /**
     * The array of valid classes for the BuySentence's subjects.
     */
    private static final Class<?>[] validSubjectClasses = { Player.class };

    /**
     * The contact methods for the Buy event.
     */
    private List<List<String>> contactMethods;

    /**
     * Creates a new BuySentence with default values (empty).
     */
    public BuySentence() {
        super(BuySentence.validObjectClasses, BuySentence.validSubjectClasses);
        this.contactMethods = new ArrayList<List<String>>();
    }

    /**
     * Creates a new BuySentence based on the given Buy {@link Tree} node.
     * @param buyToken The Buy node to build the BuySentence from.
     */
    public BuySentence(Tree<ParseToken> buyToken) {
        this();

        if (buyToken.getNode().getType().equalsIgnoreCase("buy")) {

            this.readSubObj(buyToken, "object");

            this.readSubObj(buyToken, "subject");

            this.getContactMethods(buyToken);
        }
    }

    /**
     * Gets the contact methods for the BuySentence based on the given Buy
```

```
     * {@link Tree} node.
     * @param buyToken The Buy node to get the contact methods from.
     */
    private void getContactMethods(Tree<ParseToken> buyToken) {
        MatchFunctor<String, ParseToken> typeMatcher = new TypeMatcher();
        int index = 0;
        for (Tree<ParseToken> typeNode : buyToken.findDirect("contactmethod",
                typeMatcher)) {
            ParseToken typeToken = typeNode.getNode();

            // Put in new list if OR
            if (typeToken.getLogic() == LogicOp.OR
                    && (index >= this.contactMethods.size() || this.contactMethods
                            .get(index).size() > 0)) {
                index++;
            }
            if (index >= this.contactMethods.size()) {
                this.contactMethods.add(new ArrayList<String>());
            }
            this.contactMethods.get(index).add(typeNode.getNode().getValue());
        }
    }

    @Override
    /**
     * Returns the generated sentence as a String.
     */
    public String toString() {
        if (this.eventObject.size() < 1) {
            return "< Error in BuySentence: No object to buy found >";
        }

        if (this.checkObject() && this.checkSubject()) {
            StringBuilder builder = new StringBuilder();
            Generator.appendCommaList(this.eventSubject, builder, "or");
            if (builder.length() > 0) {
                builder.append(" ");
            }
            builder.append("WTB ");
            Generator.appendCommaList(this.eventObject, builder, "or");
            if (this.contactMethods.size() > 0) {
                builder.append(", ");
                Generator.appendCommaList(this.contactMethods, builder, "or");
            }
            builder.append(".");
            return builder.toString();
        }

        else {
            return "< Error in BuySentence: Invalid object or subject found >";
        }
    }

}
```

B.3.3 SellSentence.java

```
package edu.hawaii.ctfoo.lang_generator.sentence;

import java.util.ArrayList;
import java.util.List;

import edu.hawaii.ctfoo.lang_generator.MatchFunctor;
import edu.hawaii.ctfoo.lang_generator.Generator;
import edu.hawaii.ctfoo.lang_generator.LogicOp;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.TypeMatcher;
import edu.hawaii.ctfoo.lang_generator.entity.Item;
import edu.hawaii.ctfoo.lang_generator.entity.Player;

/**
 * A {@link Sentence} where the event is selling {@link Item}s.
 *
 * @author Christopher Foo
 *
 */
public class SellSentence extends Sentence {

    /**
     * The array of valid classes for the SellSentence's objects.
     */
    private static final Class<?>[] validObjectClasses = { Item.class };

    /**
     * The array of valid classes for the SellSentence's subjects.
     */
    private static final Class<?>[] validSubjectClasses = { Player.class };

    /**
     * The contact methods of the Sell event.
     */
    private List<List<String>> contactMethods;

    /**
     * Creates a new SellSentence with the default values (empty).
     */
    public SellSentence() {
        super(SellSentence.validObjectClasses, SellSentence.validSubjectClasses);
        this.contactMethods = new ArrayList<List<String>>();
    }

    /**
     * Creates a new SellSentence based on the Sell token at the given node.
     *
     * @param sellToken
     *            The node with the sell token used to generate the
     *            SellSentence.
     */
```

```
public SellSentence(Tree<ParseToken> sellToken) {
    this();

    if (sellToken.getNode().getType().equalsIgnoreCase("sell")) {

        this.readSubObj(sellToken, "object");

        this.readSubObj(sellToken, "subject");

        this.getContactMethods(sellToken);
    }
}

/**
 * Gets the contact methods for the SellSentence from the given Sell token
 * {@link Tree} node.
 *
 * @param sellToken
 *            The {@link Tree} node of the contact methods to get.
 */
private void getContactMethods(Tree<ParseToken> sellToken) {
    MatchFunctor<String, ParseToken> typeMatcher = new TypeMatcher();
    int index = 0;
    for (Tree<ParseToken> typeNode : sellToken.findDirect("contactmethod",
            typeMatcher)) {
        ParseToken typeToken = typeNode.getNode();

        // Put in new list if OR
        if (typeToken.getLogic() == LogicOp.OR
                && (index >= this.contactMethods.size() || this.contactMethods
                        .get(index).size() > 0)) {
            index++;
        }
        if (index >= this.contactMethods.size()) {
            this.contactMethods.add(new ArrayList<String>());
        }
        this.contactMethods.get(index).add(typeNode.getNode().getValue());
    }
}

@Override
/**
 * Returns the generated sentence as a String.
 */
public String toString() {
    if (this.eventObject.size() < 1) {
        return "< Error in SellSentence: No object to sell found >";
    }

    if (this.checkObject() && this.checkSubject()) {
        StringBuilder builder = new StringBuilder();
        Generator.appendCommaList(this.eventSubject, builder, "or");
        if (builder.length() > 0) {
            builder.append(" ");
        }
```

```
            builder.append("WTS ");
            Generator.appendCommaList(this.eventObject, builder, "or");
            if (this.contactMethods.size() > 0) {
                builder.append(", ");
                Generator.appendCommaList(this.contactMethods, builder, "or");
            }
            builder.append(".");
            return builder.toString();
        }

        else {
            return "< Error in SellSentence: Invalid object or subject found >";
        }
    }

}
```

B.3.4 FindGroupSentence.java

```
package edu.hawaii.ctfoo.lang_generator.sentence;

import edu.hawaii.ctfoo.lang_generator.Generator;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.entity.Instance;
import edu.hawaii.ctfoo.lang_generator.entity.Player;

/**
 * A {@link Sentence} where the event is finding a group for an {@link Instance}
 * .
 *
 * @author Christopher
 *
 */
public class FindGroupSentence extends Sentence {

    /**
     * The array of valid classes for the FindGroupSentence's objects.
     */
    private static final Class<?>[] validObjectClasses = { Instance.class };

    /**
     * The array of valid classes for the FindGroupSentence's subjects.
     */
    private static final Class<?>[] validSubjectClasses = { Player.class };

    /**
     * Creates a new FindGroupSentence with the default values (empty);
     */
    public FindGroupSentence() {
        super(validObjectClasses, validSubjectClasses);
    }

    /**
```

```
     * Creates a new FindGroupSentence and fills it with values based on the
     * given FindGroup {@link Tree} node.
     *
     * @param findGroupToken
     *            The FindGroup Tree node used to populate the
     *            FindGroupSentence.
     */
    public FindGroupSentence(Tree<ParseToken> findGroupToken) {
        this();

        if (findGroupToken.getNode().getType().equalsIgnoreCase("findgroup")) {
            this.readSubObj(findGroupToken, "object");
            this.readSubObj(findGroupToken, "subject");
        }
    }

    @Override
    /**
     * Returns the generated sentence in String form.
     */
    public String toString() {

        if (this.eventObject.size() < 1 && this.eventSubject.size() < 1) {
            return "< Error in FindGroupSentence: No object nor subject found >";
        }

        if (this.checkObject() && this.checkSubject()) {
            StringBuilder builder = new StringBuilder();
            if (this.checkObject() && this.checkSubject()) {

                Generator.appendCommaList(this.eventSubject, builder, "or");
                if (builder.length() > 0) {
                    builder.append(" ");
                }
                builder.append("LFG");
                if (this.eventObject.size() > 0) {
                    builder.append(" for ");
                    Generator.appendCommaList(this.eventObject, builder, "or");
                }
                builder.append(".");
            }
            return builder.toString();
        } else {
            return "< Error in FindGroupSentence: Invalid subject or object found >";
        }
    }
}
```

B.3.5 FindMoreSentence.java

```
package edu.hawaii.ctfoo.lang_generator.sentence;

import java.util.ArrayList;
import java.util.List;
```

```java
import edu.hawaii.ctfoo.lang_generator.MatchFunctor;
import edu.hawaii.ctfoo.lang_generator.Generator;
import edu.hawaii.ctfoo.lang_generator.LogicOp;
import edu.hawaii.ctfoo.lang_generator.ParseToken;
import edu.hawaii.ctfoo.lang_generator.Tree;
import edu.hawaii.ctfoo.lang_generator.TypeMatcher;
import edu.hawaii.ctfoo.lang_generator.entity.Entity;
import edu.hawaii.ctfoo.lang_generator.entity.Instance;
import edu.hawaii.ctfoo.lang_generator.entity.Player;

/**
 * A {@link Sentence} where the event is looking for more members.
 *
 * @author Christopher Foo
 *
 */
public class FindMoreSentence extends Sentence {

    /**
     * The array of classes that are valid objects in the FindMoreSentence.
     */
    private static final Class<?>[] validObjectClasses = { Player.class };

    /**
     * The array of classes that are valid subjects in the FindMoreSentence.
     */
    private static final Class<?>[] validSubjectClasses = {};

    /**
     * The {@link Instance}s that more members are being sought for.
     */
    private List<List<Instance>> instances;

    /**
     * Creates a new FindMoreSentence with default values (empty).
     */
    public FindMoreSentence() {
        super(validObjectClasses, validSubjectClasses);
        this.instances = new ArrayList<List<Instance>>();
    }

    /**
     * Creates a new FindMoreSentence filled with values based on the given
     * FindMore token {@link Tree} node.
     *
     * @param findMoreToken
     *            The FindMore Tree node used to fill the FindMoreSentence.
     */
    public FindMoreSentence(Tree<ParseToken> findMoreToken) {
        this();

        if (findMoreToken.getNode().getType().equalsIgnoreCase("findmore")) {

            this.readSubObj(findMoreToken, "object");
```

```
        this.readSubObj(findMoreToken, "subject");
        this.getInstances(findMoreToken);
    }
}

/**
 * Gets the {@link Instance}s that members are being sought for in the
 * FindMore event.
 *
 * @param findMoreToken
 *            The FindMore {@link Tree} node describing this
 *            FindMoreSentence.
 */
private void getInstances(Tree<ParseToken> findMoreToken) {
    MatchFunctor<String, ParseToken> typeMatcher = new TypeMatcher();
    int index = 0;
    for (Tree<ParseToken> typeNode : findMoreToken.findDirect("instance",
            typeMatcher)) {
        ParseToken typeToken = typeNode.getNode();

        // Put in new list if OR
        if (typeToken.getLogic() == LogicOp.OR
                && (index >= this.instances.size() || this.instances.get(
                        index).size() > 0)) {
            index++;
        }
        if (index >= this.instances.size()) {
            this.instances.add(new ArrayList<Instance>());
        }
        this.instances.get(index).add(new Instance(typeNode));
    }
}

@Override
/**
 * Returns the generated sentence as a String.
 */
public String toString() {
    if (this.eventObject.size() < 1 && this.eventSubject.size() < 1
            && this.instances.size() < 1) {
        return "< Error in FindMoreSentence: No subject or object found >";
    }

    if (this.checkObject() && this.checkSubject()) {
        StringBuilder builder = new StringBuilder();
        Generator.appendCommaList(this.eventSubject, builder, "or");
        builder.append(" LF");
        int numMembersSmall = 0;
        int numMembersBig = 0;
        for (List<Entity> outerList : this.eventObject) {
            int numMembersTemp = 0;
            for (Entity object : outerList) {
                if (object instanceof Player) {
                    numMembersTemp += ((Player) object).getQuantity();
                }
```

```
                }

                // If first set, initialize
                if (numMembersSmall == 0) {
                    numMembersSmall = numMembersTemp;
                    numMembersBig = numMembersTemp;
                }

                // We found the number of members before
                else {
                    if (numMembersTemp < numMembersSmall) {
                        numMembersSmall = numMembersTemp;
                    } else if (numMembersTemp > numMembersBig) {
                        numMembersBig = numMembersTemp;
                    }
                }
            }

            if (numMembersSmall > 0) {
                if (numMembersSmall == numMembersBig) {
                    builder.append(numMembersSmall);
                } else {
                    builder.append(numMembersSmall + "-" + numMembersBig);
                }
            }

            builder.append("M ");
            Generator.appendCommaList(this.eventObject, builder, "or");
            if (this.instances.size() > 0) {
                if (this.eventObject.size() > 0) {
                    builder.append(" ");
                }
                builder.append("for ");
                Generator.appendCommaList(this.instances, builder, "or");
            }
            builder.append(".");
            return builder.toString().trim();
        }

        else {
            return "< Error in FindMoreSentence: Invalid subject or object found >";
        }
    }
}
```

Appendix C: Context Free Grammar Rules

Note: < > indicates non-terminal, [ ] indicates optional

C.1 Sentence Level Rules

<BuySentence> → [<Players>] WTB <Items>[, <ContactMethods>].

<SellSentence> → [<Players>] WTS <Items>[, <ContactMethods>].

<FindGroupSentence> → <Players> LFG [for <Instances>]. | [<Players>] LFG for <Instances>.

<FindMoreSentence> → [<Players>] LF[<Amount>]M <Players> [for <Instances>].
Note: <Amount> is calculated from the second <Players>

C.2 List Rules

<Entities> → <Entity> | <Entity> and <Entity> | <Entity> or <Entity> | <Entity>, <MoreEntities>

<MoreEntities> → <Entity>, <MoreEntities> | and <Entity> | or <Entity> | <Entities>, <MoreEntities> | and <Entities> | or <Entities>

Note: Generic forms. <Entity>, <Entities>, and <MoreEntities> may be replaced with any non-terminal on the left-hand side of the Entity rules below.

C.3 Entity Rules

Note: The non-terminals on the right-hand side at this level are terminated with information from the parse.

<Player> → [<Quantity>] [<Level>] [<ItemLevel>] [<Race>] [<Specializations>] <Class> | [<Quantity>] [<Level>] [<ItemLevel>] [<Specializations>] <Role>

<Instance> → [<Difficulty>] [<Mode>] <Name>

<Item> → [<Quantity>] <Name> [for <Value>] |
[<Rarity>] [<Quantity>] [Item Level <Level>] <Type> [for <Value>]

<MoneyAmount> → <Denomination> <Currency>