Mazzei-Parte 1

LEZIONI TLN

Tecnologie del Linguaggio Naturale

Università di Torino

Anno Accademico 2019/20

Sommario

1 Documentation

1.1 Introduction

SimpleNLG is a simple Java API designed to facilitate the generation of Natural Language. It was originally developed at the University of Aberdeen's Department of Computing Science.

SimpleNLG is intended to function as a "realisation engine" for Natural Language Generation architectures, and has been used successfully in a number of projects, including the BabyTalk and BabyTalk-Family Projects.

For details of its design and examples of its use, please refer to the following publication:

- A. Gatt and E. Reiter. (2009). SimpleNLG: A realisation engine for practical applications. Proceedings of the 12th European Workshop on Natural Language Generation (ENLG-09). The above paper is also the best academic citation for simplenlg.

For other papers and presentations about simplenlg, please see the Papers and Presentations page.

The tutorial provides information about the simpleNLG library, how to set it up and how to use it. The following pages assume that you have a basic knowledge of java and object oriented programming. For example, you will need to understand what a class and an instance of a class are.

1.1.1 Version history and hosting

SimpleNLG is currently hosted on GitHub.

Earlier versions of SimpleNLG can be are no longer being maintained. Please note that earlier versions of simplenlg have different licensing, in particular versions before V4.0 cannot be used commercially.

1.1.2 Summary of features

SimpleNLG handles the following:

- Lexicon/morphology system: The default lexicon computes inflected forms (morphological realisation). We believe this has fair coverage. Better coverage can be obtained by using the NIH Specialist Lexicon (which is supported by simplenlg).
- Realiser: Generates texts from a syntactic form. Grammatical coverage is very limited compared to tools such as KPML and FUF/SURGE, but we believe it is adequate for many NLG tasks.
  Microplanning: Currently just simple aggregation, hopefully will grow over time.

1.2 What is SimpleNLG

SimpleNLG can be used to help you write a program which generates grammatically correct English sentences. It's a library (not an application), written in Java, which performs simple and useful tasks that are necessary for natural language generation (NLG).

SimpleNLG is distributed as a stand-alone jar file, available under the Downloads tab. The jar file contains all the classes you'll need, as well as a lexicon (discussed later). If you are interested in the source, it is available here.

Because it's a library, you will need to write your own Java program which makes use of SimpleNLG classes. These classes will allow you to specify the subject of a sentence ('my dog'), the exact verb you want to appear in the sentence ('chase'), the object ('George'), and additional complements ('because George looked funny'). You can also use SimpleNLG methods to indicate, for example, that you would like the verb to be in the past tense and expressed in the progressive form ('was chasing'). If this is already confusing, don't worry -- this tutorial will help you with all of that.

Once you have stipulated what the content of your sentence will be and expressed this information in SimpleNLG terms, SimpleNLG can assemble the parts of your sentence into a grammatical form and output the result. In our example, the resulting output would be "My dog was chasing George because George looked funny". Here, SimpleNLG has:

1. Organized all the different parts into the correct order for English.
2. Capitalized the first letter of the sentence.
3. Added the auxiliary 'was' and made it agree with the subject.
4. Added '-ing' to the end of the verb (because the progressive aspect of the verb is desired).
5. Put all the words together in a grammatical form.
6. Inserted the appropriate whitespace between the words of the sentence.
7. Put a period at the end of the sentence.

As you can see, SimpleNLG will not choose particular words for you: you will need to specify the words you want to appear in the output and their parts of speech. What SimpleNLG's library of classes will do for you is create a grammatically correct sentence from the parts of speech you have provided it with. SimpleNLG automates some of the more mundane tasks that all natural language generation (NLG) systems need to perform. (For more information on NLG, see Appendix A). Tasks such as:

Orthography:

- Inserting appropriate whitespace in sentences and paragraphs.
- Absorbing punctuation – for example, generating the sentence "He lives in Washington D.C." instead of "He lives in Washington D.C.." (i.e., the sentence ends with a single period rather than two).
- Pouring – that is, inserting line breaks between words (rather than in the middle of a word) in order to fit text into rows of, for example, 80 characters (or whatever length you choose).
- Formatting lists such as: "apples, pears and oranges."

Morphology:

- Handling inflected forms – that is, modifying/marking a word/lexeme to reflect grammatical information such as gender, tense, number or person.

Simple Grammar:

- Ensuring grammatical correctness by, among other things, enforcing noun-verb agreement [1].
- Creating well-formed verb groups (i.e., verb plus auxiliaries) such as "does not like".
- Allowing the user to define parts of a sentence or phrase and having simplenlg gather those parts together into an appropriate syntactic structure.

For those familiar with the terminology of natural language generation (NLG), SimpleNLG is a realiser for a simple grammar. We hope that SimpleNLG will eventually provide simple algorithms for not only realization but all of microplanning as well. As its functionality expands over time, components such as microplanning will be added as self-contained modules: self-contained, in order to allow students and researchers use of parts of the library they want, with the freedom to extend or replace other modules with their own implementations.

1.3 Who uses SimpleNLG

Some of the people who will want to use the SimpleNLG library will be:

- Researchers who are concentrating on their own implementations of document planning or microplanning and who don’t want to be bothered with the automatic and mundane tasks of realisation.
- Students who want to learn more about natural language generation.
- Anyone who wants to write programs that can generate English sentences.

1.4 Getting started

t's important to note that SimpleNLG is a library [1], not an application! This means you cannot run it as a Java program – it does not have a "main" method. You need to create your own Java program which makes use of simplenlg classes and methods.

To enable your program to make use of the simplenlg library, you'll need to:

- Download the simplenlg zip file from the Download tab.
- Extract then add simplenlg's jar file to your classpath. (For instructions on how to do this in Eclipse, see Appendix B).
- Create a new Java class which has the main method in it. In this example, let's call the new class TestMain.
- At the top of that class, put in the following import statements:

```
import simplenlg.framework.*;
import simplenlg.lexicon.*;
import simplenlg.realiser.english.*;
import simplenlg.phrasespec.*;
import simplenlg.features.*;
```

Create a SimpleNLG lexicon, NLGFactory, and realiser using the following statements:

```
Lexicon lexicon = Lexicon.getDefaultLexicon();
NLGFactory nlgFactory = new NLGFactory(lexicon);
Realiser realiser = new Realiser(lexicon);
```

Following these steps, you should have code that looks like the following:

```
import simplenlg.framework.*;
import simplenlg.lexicon.*;
import simplenlg.realiser.english.*;
import simplenlg.phrasespec.*;
import simplenlg.features.*;


public class TestMain {

        public static void main(String[] args) {
                Lexicon lexicon = Lexicon.getDefaultLexicon();
                NLGFactory nlgFactory = new NLGFactory(lexicon);
                Realiser realiser = new Realiser(lexicon);

        }

}
```

Figure 1: A Java class which is ready to make use of the simplenlg library.

You're now ready to make use of simplenlg to generate sentences!

→ For further examples on ways to use simplenlg, take a look at the java files in testsrc. A stand-alone example is provided in testsrc/StandAloneExample.java.

Generating the simplest kind of phrase in SimpleNLG

Let's create the simplest kind of sentence allowed in SimpleNLG: canned text, i.e., a string which we'd like output to the screen as is. For instance, let's say we are writing a program which takes input from users and generates a different paragraph depending on the various inputs. But let's say that no matter what, we always want the first line of the paragraph to be "My dog is happy" because we feel everyone should share in the good news. The SimpleNLG code to do this is:

```
NLGElement s1 = nlgFactory.createSentence("my dog is happy");
```

We now need to use the Realiser to output the string:

```
String output = realiser.realiseSentence(s1);
System.out.println(output);
```

It's important to note that you only need to create the Lexicon, NLGFactory, and Realiser once within your program; you don't need to create them for every sentence you generate. So a good idea is to create these at the start of your program and use them over the lifetime of the program run.

- **The main steps involved in generating a more complicated sentence**

The last example was the simplest way to create a sentence, and as you can see, it requires you to do most of the work. Now let's take a look at how to get SimpleNLG to do most of the work. Keep in mind that SimpleNLG is pretty flexible in this way, and there are several other ways to generate sentences – many of the things that you think of will probably work as well.

At the most fine-grained level, SimpleNLG understands what a sentence is using a class called SPhraseSpec. This is accessible through the NLGFactory, using the createClause method. The ideas of 'verb phrase', 'noun phrase', 'prepositional phrase', 'adjective phrase', and 'adverb phrase' are are also all accessible through NLGFactory, using createVerbPhrase, createNounPhrase, createPrepositionPhrase, etc. These return similarly named classes – VPPhraseSpec, NPPhraseSpec, PPPhraseSpec, AdjPhraseSpec, and AdvPhraseSpec.

In order to build a sentence using these SimpleNLG concepts or classes, you will normally follow these steps (all of this will be further explained in Sections V and on):

- Create an instance of NLGFactory.
- Create a placeholder for the sentence using NLGFactory's createClause method. (This returns an SPhraseSpec instance.)
- Create a verb, subject, and object using NLGFactory's createVerbPhrase and createNounPhrase methods. (These return VPPhraseSpec and NPPhraseSpec instances.)
- If you want, create prepositional phrases, adjective phrases, and adverb phrases using NLGFactory's createPrepositionPhrase, createAdjectivePhrase, and createAdverbPhrase methods. (These return PPPhraseSpec, AdjPhraseSpec and AdvPhraseSpec instances.)
- Indicate what role these various parts of speech will play in the desired sentence. For example, specify that you want a particular noun phrase to be the subject of the sentence, and some other noun phrase to be the object, using setSubject and setObject. Specify the verb using setVerb, and the complement (e.g., the prepositional phrase) using addComplement.

- Create a SimpleNLG object called the Realiser.
- Ask the Realiser to 'realise' or transform the SPhraseSpec instance into a syntactically correct string.

You now have a string which is a grammatical English phrase or sentence and it can be treated like any other Java string. For instance, you can manipulate it further or print it out using the Java method System.out.println.

Note that this is the most detailed approach: You can actually just use setSubject, setVerb, etc., by passing these methods simple strings as arguments. Unless you're doing more complicated things, first specifying that a subject is an NPPhraseSpec or that a verb is a VPPhraseSpec is not even necessary! See Section V for an example of actual Java code used to generate a sentence.

Below is quick breakdown of the main parts of speech that simplenlg can handle. The rest of the tutorial will discuss each of these parts of speech in more detail.

Part of Speech	Phrase Type	Examples	Method
Subject	Noun Phrase	"the boy"	setSubject("blah")
Verb	Verb Phrase	"gave"	setVerb("blah")
Object	Noun Phrase	"the dog"	setObject("blah")
Indirect Object	Noun Phrase	"a present"	setIndirectObject("blah")
Complement	Preposition Phrase	"in the park"	addComplement("blah")
	that-clause[2]	"that Sarah sees John"	
	Adjective Phrase	"delighted to meet you"	
	Adverb Phrase	"very quickly"	
Modifier	that-clause	"the girl *that I knew*"	addModifier("blah")
	Adjective Phrase	"*pretty* flower"	
	Adverb Phrase[3]	"*quickly* chases"	
Article	Determiner Phrase	"*the* boy"	setDeterminer("a")

Table 1: The parts of speech that *simplenlg* can handle. Both modifiers and articles are added to other phrases; the rest are added to the sentence.

---

[1] A library or API is a collection of methods/functions that people can make use of in their programs. This saves programmers from having to write that code themselves.

[2] This is actually a Complementiser Phrase, which is not currently implemented in simplenlg. Instead, such phrases can just be written as strings, then specified as complements.

[3] This may also be added as a modifier to the full sentence, and will be interpreted as modifying the main verb.

1.5 Lexicon

Like other natural language processing systems, SimpleNLG needs information about words; this is called a Lexicon. Simplenlg comes with a simple lexicon built into the system, which is accessed via:

```
Lexicon lexicon = Lexicon.getDefaultLexicon();
```

SimpleNLG can also use the 300MB NIH Specialist lexicon, which has outstanding coverage of medical terminology as well as excellent coverage of everyday English. For information on setting up this lexicon, please see Appendix C.

You can also create your own lexicon if you wish. The easiest way to do this is probably to edit default-lexicon.xml (the default lexicon file), this is in res/default-lexicon.xml in the simplenlg zip file. If your new lexicon is called my-lexicon.xml, and is saved in your current working directory, you can access it as follows:

```
Lexicon lexicon = new XMLLexicon("my-lexicon.xml");
```

To access a lexicon outside of the current working directory, provide the full path name (e.g., "/home/staff/lexicons/my-lexicon.xml", "C:\lexicons\my-lexicon.xml").[1]

Once we have a lexicon, we can create an NLGFactory (object which creates simplenlg structures) and a realiser (object which transforms simplenlg structures into text), as follows:

```
NLGFactory nlgFactory = new NLGFactory(lexicon);
Realiser realiser = new Realiser(lexicon);
```

→ For more examples, look at testsrc/MultipleLexiconTest.java and testsrc/NIHDBLexiconTest.java.

---

[1] Note that in SimpleNLG V4, there are no lexicon methods to directly get inflected variants of a word; in other words, there is no equivalent in V4 of the SimpleNLG V3 getPlural(), getPastParticiple(), etc. methods. It is possible in V4 to compute inflected variants of words, but the process is more complicated: basically we need to create an InflectedWordElement around the base form, add appropriate features to this InflectedWordElement, and then realise it.

1.6 Modifiers vs complements

As shown in Table 1, there are several different kinds of phrase in SimpleNLG: noun phrases (which are represented by the Java class NPPhraseSpec), clauses or full sentences (which are represented by SPhraseSpec), prepositional phrases (represented by the class PPPhraseSpec), adjective phrases and adverb phrases. Adjective phrases and adverb phrases can be generated using the SimpleNLG concepts of modifier (when they modify a specific word/phrase) and complement (when they should occur after the verb).

SimpleNLG in fact distinguishes between three types of modifiers: front modifiers (which go at the beginning of a phrase), pre-modifiers (which go immediately before the main noun or verb in a phrase), and post-modifiers (which go at the end of a phrase). You can directly specify where a modifier goes by using addFrontModifier(), addPremodifier(), or addPostmodifier(). If you use the more general addModifier(), then SimpleNLG will decide where to place your modifier.

Pre- and post-modifiers are allowed in all types of phrases. Front modifiers can only be specified for SPhraseSpec.

1.7 Generating a simple sentence

In the sample Java class TestMain shown above, we have the statements:

```
import simplenlg.framework.*;
import simplenlg.lexicon.*;
import simplenlg.realiser.english.*;
import simplenlg.phrasespec.*;
import simplenlg.features.*;
```

These classes allow you to specify the parts of speech of a sentence and to perform various operations on them. It's important to note that SimpleNLG provides only a simple grammar: its notions of a sentence, noun phrase, etc., are very basic and are by no means representative of the incredibly varied and complicated grammar of the English language.

Let's see how we would define and combine various parts of speech to generate a simple sentence such as "Mary chases the monkey". As discussed in Section III, we'll make use of the SimpleNLG construct SPhraseSpec, which allows us to define a sentence or a clause in terms of its syntactic constituents. This is useful because it allows us to hand different parts of a clause to SimpleNLG, in no particular order, and SimpleNLG will assemble those parts into the appropriate grammatical structure.

```
SPhraseSpec p = nlgFactory.createClause();
p.setSubject("Mary");
p.setVerb("chase");
p.setObject("the monkey");
```

The above set of calls to SimpleNLG defines the components of the sentence we wish to construct: we have specified a subject, a verb and an object for our sentence. Now, all that remains is to use the Realiser, which will take these different components of the sentence, combine them, and realise the text to make the result syntactically and morphologically correct:

```
String output2 = realiser.realiseSentence(p); // Realiser created earlier.
System.out.println(output2);
```

The resulting output is:

```
Mary chases the monkey.
```

When parts of speech are defined and assembled into an instance of the SPhraseSpec class, methods associated with that class such as setSubject, setVerb and setObject, assemble the parts of speech by obeying the simple grammar embodied in SimpleNLG [1].

→ For more examples on clauses, look at testsrc/ClauseTest.java.

---

[1] And, as we will see later, rules of grammar will have also been enforced in building up the smaller constituents of the sentence (such as NPPhraseSpec and PPPhraseSpec) to ensure they are well-formed. Thus, the rules of grammar which SimpleNLG implements are not defined within a single module of the SimpleNLG code but instead are spread throughout the various class definitions.

1.8 Verbs

Verbs should be specified in infinitive ("to XXX") form. However, as a convenience, simplenlg will usually accept inflected forms of verbs as well. For example:

```
p.setVerb("is");
```

is equivalent to

```
p.setVerb("be");
```

You can specify particles (prepositions which accompany a verb) by writing the following:

```
p.setVerb("pick up");
p.setVerb("put down");
```

Verbs in SimpleNLG can have one of three different tenses: past, present and future. Let's say we've written the following SimpleNLG code which yields the sentence "Mary chases the monkey.":

```
SPhraseSpec p = nlgFactory.createClause();
p.setSubject("Mary");
p.setVerb("chase");
p.setObject("the monkey");
```

In order to set this in the past, we would add the line:

```
p.setFeature(Feature.TENSE, Tense.PAST);
```

thus rendering the sentence:

```
Mary chased the monkey.
```

If Mary is instead busy with other things and forced to postpone her exercise, we could write:

```
p.setFeature(Feature.TENSE, Tense.FUTURE);
```

yielding the sentence:

```
Mary will chase the monkey.
```

Negation If negated is set to true, the negative form of the sentence is produced. For example adding the following line to the previous:

```
p.setFeature(Feature.NEGATED, true);
```

will change the resulting sentence to:

```
Mary will not chase the monkey.
```

Questions Simplenlg can generate simple yes/no questions. For example:

```
p.setSubject("Mary");
p.setVerb("chase");
p.setObject("the monkey");
p.setFeature(Feature.INTERROGATIVE_TYPE, InterrogativeType.YES_NO);
```

will generate:

```
Does Mary chase the monkey?
```

Simplenlg can also generate simple WH questions. For example:

```
p.setSubject("Mary");
p.setVerb("chase");
p.setFeature(Feature.INTERROGATIVE_TYPE, InterrogativeType.WHO_OBJECT);
```

will generate:

```
Who does Mary chase?
```

→ For more examples on questions, look at testsrc/InterrogativeTest.java.

Features TENSE, NEGATED, and INTERROGATIVE_TYPE are examples of features which can be set on a SPhraseSpec. Many other features are also allowed, including MODAL, PASSIVE, PERFECT, and PROGRESSIVE. Detailed information about allowable features are given in the simpleNLG API documentation.

→ For more examples on verbs, look at testsrc/VerbPhraseTest.java.

1.9 What are complements

As far as SimpleNLG is concerned, a complement is anything that comes after the verb. When you label something as a complement and hand it to simplenlg to be realized, SimpleNLG will place it, no matter what it is, after the verb [1]. If you've specified an object, it will place it after both the verb and the object.

Examples of complements are italicized in the sentences below:

1. Mary is *happy*.
2. Mary wrote the letter *quickly*.
3. Mary just realized *that her holidays are over*.

The *italicized* words and phrases in the examples above are all different parts of speech. In example #1, it's an adjective phrase; in example #2, it's an adverb; and it's a 'that-clause' (complementiser phrase) in example #3. But from simplenlg's point of view, the *italicized* bits all have one thing in common: they are complements and appear after the verb. Although it has an understanding of subjects, verbs, and objects, simplenlg has a very limited concept of adjective phrases that follow verbs, adverbial phrases, that-clauses, or other parts of speech that can appear after a verb. But it does understand the concept of a complement, and because of this, phrases that appear after a verb can be generated using the simplenlg library. As can be seen in Table 2, complements encompass a variety of phrase types. (For a full listing of Parts of Speech simplenlg can handle, see Table 1.)

Part of Speech	Phrase Type	Examples
Complement	Prepositional Phrase	"in the park"
	that-clause	"that Sarah sees John"
	Adjective Phrase	"delighted to meet you"
	Adverb Phrase	"very quickly"

Table 2: Prepositional phrases, that-clauses, adjective phrases and adverbial phrases that occur at the end of a sentence fall under the header 'complement'.

Complements can be added to sentences via the addComplement method. For example, given our subject, object, and verb:

```
p.setSubject("Mary");
p.setVerb("chase");
p.setObject("the monkey");
```

We can then add any kind of complement:

```
p.addComplement("very quickly"); // Adverb phrase, passed as a string
p.addComplement("despite her exhaustion"); // Prepositional phrase, string
```

Which will generate:

```
Mary chases the monkey very quickly despite her exhaustion.
```

Note that only things of type SPhraseSpec can take a complement. Nouns and verbs take modifiers, which we discuss next.

→ For more examples on complements, look at testsrc/FPTest.java.

[1] Even if you label a nonsense string like "shabadoo" as a complement, simplenlg will happily add it after the verb.

1.10 Adding adjectives via modifier

We know for a fact that Mary is an good runner, so we'd like to assign Mary a suitable adjective, like 'fast'. This can be generated by the simplenlg library using the concept of modifier.

To deem Mary 'fast', however, you will no longer want to refer to her simply as the 'subject' of the sentence. Instead, let's also define her name as a noun phrase (which it is). In that way, we can ascribe the adjective 'fast' to Mary (which she certainly is) by means of the modifier function.

```
NPPhraseSpec subject = nlgFactory.createNounPhrase("Mary");
```

While we're at it, let's also define the object as a noun phrase and the verb as a verb phrase. This will allow us to do some fancier stuff later on, like adding a modifier to each.

```
NPPhraseSpec object = nlgFactory.createNounPhrase("the monkey");  [1]
VPPhraseSpec verb = nlgFactory.createVerbPhrase("chase");
```

Now, we can apply the adjective 'fast' to Mary by writing:

```
subject.addModifier("fast");
```

Next, we set the subject, object, and verb on the SPhraseSpec p that we defined earlier:

```
p.setSubject(subject);
p.setObject(object);
p.setVerb(verb);

String output3 = realiser.realiseSentence(p); // Realiser created earlier.
System.out.println(output3);
```

The output will be:

```
Fast Mary chases the monkey.
```

Similarly, we can let the world know that Mary, true to her nature, is chasing the monkey quickly. The adverb 'quickly' may also be added using the addModifier function, but this time it's the verb that's being modified:

```
verb.addModifier("quickly");
```

The output will be:

```
Fast Mary quickly chases the monkey.
```

→ For more examples on noun phrases, look at testsrc/NounPhraseTest.java.

→ For more examples on modifiers, look at testsrc/AdjectivePhraseTest.java.

[1] Note that we can also construct the noun phrase "the monkey" in the following way:

```
    NPPhraseSpec object = nlgFactory.createNounPhrase("monkey");
    object.setDeterminer("the"); // currently this is 'deprecated'!
```

1.11 Adding multiple subjects, objects and complements

An SPhraseSpec can have multiple subjects, objects and complements but not multiple verbs (although a future version of SimpleNLG might include this functionality). This is done with the CoordinatedPhraseElement class. Let's say you also have a giraffe that wants to chase the poor monkey. To place your giraffe in the fray, you would write:

```
    NPPhraseSpec subject1 = nlgFactory.createNounPhrase("Mary");
    NPPhraseSpec subject2 = nlgFactory.createNounPhrase("your", "giraffe");

    CoordinatedPhraseElement subj = nlgFactory.createCoordinatedPhrase(subject1,
subject2);
              // may revert to nlgFactory.createCoordinatedPhrase( subject1,
subject2 ) ;
    p.setSubject(subj);
```

The resulting output is:

```
    Mary and your giraffe chase the monkey.
```

SimpleNLG has automatically added the conjunction 'and' and has changed the ending of the verb so that it agrees with the multiple subjects of the sentence.

Similarly, you can have multiple objects and complements in an SPhraseSpec. Let's suppose Mary and your giraffe have found more people to terrorize in what's turning out to be a growing parade of horror. Instead of p.setObject("the monkey"), you would write:

```
    NPPhraseSpec object1 = nlgFactory.createNounPhrase("the monkey");
    NPPhraseSpec object2 = nlgFactory.createNounPhrase("George");

    CoordinatedPhraseElement obj = nlgFactory.createCoordinatedPhrase(object1,
object2);
              // may revert to nlgFactory.createCoordinatedPhrase( subject1,
subject2 ) ;
    obj.addCoordinate("Martha");
    p.setObject(obj);
```

The resulting output will be:

```
    Mary and your giraffe chase the monkey, George and Martha.
```

If Mary and the devious giraffe run by so quickly that you can't see who they're chasing, you can change the conjunction on the coordinated elements:

```
    obj.setFeature(Feature.CONJUNCTION, "or");
```

The resulting output will be:

```
    Mary and your giraffe chase the monkey, George or  Martha.
```

As with many methods in SimpleNLG, the createCoordinatedPhrase method can take all kinds of arguments – NPPhraseSpec, PPPhraseSpec, or even simple strings.

→ For more examples on coordination, look at testsrc/ClauseAggregationTest.java

1.12 Prepositional phrases

Our sentence is getting rather crowded with people and animals. So let's return to the pristine simplicity of Mary chasing the monkey. But let's give the heart-pounding action a setting: "Mary chases the monkey **in the park**".

The phrase "in the park" is a prepositional phrase and there are a number of ways we can create it using simplenlg. The simplest way would be to pass the string "in the park" to the addComplement method:

```
SPhraseSpec p = nlgFactory.createClause("Mary", "chase", "the monkey");
p.addComplement("in the park");
```

A more sophisticated way of creating this prepositional phrase, however, would be to specify the parts of the prepositional phrase – the preposition, determiner, noun phrase – and combine them:

```
NPPhraseSpec place = nlgFactory.createNounPhrase("park");
place.setDeterminer("the");
PPPhraseSpec pp = nlgFactory.createPrepositionPhrase();
pp.addComplement(place);
pp.setPreposition("in");
```

We then add the prepositional phrase as a complement to the 'Mary chases the monkey' sentence.

```
p.addComplement(pp);
```

The table below shows three different ways of creating the prepositional phrase "in the park". As you can start to see, there are often several ways to realise sentences using simplenlg. The best way to do it depends on the needs of your system.

SPhraseSpec p = nlgFactory.createClause("Mary", "chase", "the monkey");	
Simple way:	**p.addComplement("in the park");**
Semi-sophisticated way:	**PPPhraseSpec pp = nlgFactory.createPrepositionPhrase("in", "the park"); p.addComplement(pp);**

Sophisticated way:	**NPPhraseSpec place = nlgFactory.createNounPhrase("park");
	place.setDeterminer("the");
	PPPhraseSpec pp = nlgFactory.createPrepositionPhrase();
	pp.addComplement(place);
	pp.setPreposition("in");
	p.addComplement(pp);**

String output = realiser.realiseSentence(p); // Realiser created earlier.
System.out.println(output);	

Mary chases the monkey in the park.	

Table 3: Three ways of adding the prepositional phrase 'in the park' to a sentence.

The third, sophisticated method requires much more code than the other two ways. So why then would we ever choose the third method?

The main reason is that the third method allows you to add pieces to a phrase or sentence with much greater ease. We have to remind ourselves that simplenlg will normally be used in a larger program which chooses the content of a sentence – and that content will likely be determined in a piecemeal fashion. It's much easier to have simplenlg add a word or clause to a phrase which has been defined in a modular way (i.e., parts of the sentence are divided into chunks and labeled) rather than having to add new information to a monolithic string whose parts are not differentiated. For example, if we wanted to describe the park as 'leafy', and we had used the 3rd method to define our sentence, all we would need to do is write the following code:

```
place.addPreModifier("leafy");
```

Had we chosen the first or second methods, however, adding the adjective 'leafy' to the string 'park' would be a major hassle. Among other things, you would have to write code which could:

- Find where to insert the new word in the string. In most cases this would require parsing the string, which is no simple task!
- Break that string into pieces to allow the insertion.
- Insert the word.
- Determine whether that insertion requires changing the other bits of string.
- Put the pieces of string back together in a grammatical way.

In other words, you would have to write a realiser like SimpleNLG!

So why, given the major drawback stated above, would we ever choose to define a sentence using methods #1 and #2? Because sometimes we simply want to generate canned text, i.e., text that we know won't need to be enlarged or changed and which we simply want output as-is. If we know that we won't be changing a phrase (such as 'in the park'), then it makes sense to treat it as a uniform entity the way the first two methods do.

→ For more examples of prepositional phrases, look at

testsrc/PrepositionalPhraseTest.java.

1.13 Different ways of specifying a phrase

We have now covered a bunch the basic functionality of SimpleNLG. One thing that this tutorial has outlined is the fact that there are numerous ways of specifying a phrase.

In order to make this a bit more explicit, the table below shows some of the ways we can create the sentence 'Mary chases the monkey'. You can define all the components of the phrase when you create an instance of it (as in example #1). Or you can create the instance first and then add the components one at a time (as in example #2). Alternatively, the components of a sentence can themselves be phrases (as in example #3). Or you can have a combination of all these various syntaxes (as in examples 4-5).

	Lexicon lexicon = Lexicon.getDefaultLexicon(); NLGFactory nlgFactory = new NLGFactory(lexicon); Realiser realiser = new Realiser(lexicon);
1.	**SPhraseSpec p = nlgFactory.createClause("Mary", "chase", "the monkey");**
2.	**SPhraseSpec p = nlgFactory.createClause();** **p.setSubject("Mary");** **p.setVerb("chase");** **p.setObject("the monkey");**
3.	**NPPhraseSpec subj = nlgFactory.createNounPhrase("Mary");** **NPPhraseSpec obj = nlgFactory.createNounPhrase("monkey");** **obj.setDeterminer("the");** **VPPhraseSpec verb = nlgFactory.createVerbPhrase("chase");** **SPhraseSpec p = nlgFactory.createClause(subj, verb, obj);**
4.	**SPhraseSpec p = nlgFactory.createClause();** **NPPhraseSpec obj = nlgFactory.createNounPhrase("the monkey");** **p.setSubject("Mary");** **p.setVerb("chase");** **p.setObject(obj);**
5.	**SPhraseSpec p = nlgFactory.createClause("Mary", "chase",** **nlgFactory.createNounPhrase("the monkey"));**

	String output = realiser.realiseSentence(p); // Realiser created earlier.
	System.out.println(output);

Table 4: Different ways of creating the sentence "Mary chases the monkey".

Note: Currently .setDeterminer("the") is set as deprecated!

1.14 Generating a sentence with multiple clauses

1.14.1 Phrases joined by a conjunction

One way of generating a sentence with multiple clauses is to use the SimpleNLG class CoordinatedPhraseElement.

```
SPhraseSpec s1 = nlgFactory.createClause("my cat", "like", "fish");
SPhraseSpec s2 = nlgFactory.createClause("my dog", "like", "big bones");
SPhraseSpec s3 = nlgFactory.createClause("my horse", "like", "grass");

CoordinatedPhraseElement c = nlgFactory.createCoordinatedPhrase();
c.addCoordinate(s1);
c.addCoordinate(s2);
c.addCoordinate(s3);
```

The CoordinatedPhraseElement c object can then be realised as a sentence:

```
String output = realiser.realiseSentence(c);
System.out.println(output);
```

If you do not supply a conjunction using the method setConjunction, the conjunction 'and' will automatically be used because it is the default. In this case, the resulting sentence would be:

```
My cat likes fish, my dog likes big bones and my horse likes grass.
```

1.14.2 Subordinate clauses

Subordinate clauses can be added to the main clause using the addComplement method, where the kind of complementiser ("because", "while", etc.) to be used is set using the setFeature method.

```
SPhraseSpec p = nlgFactory.createClause("I", "be", "happy");
SPhraseSpec q = nlgFactory.createClause("I", "eat", "fish");

q.setFeature(Feature.COMPLEMENTISER, "because");
q.setFeature(Feature.TENSE, Tense.PAST);
p.addComplement(q);

String output4 = realiser.realiseSentence(p);  //Realiser created earlier
System.out.println(output4);
```

The output is:

```
I am happy because I ate fish.
```