A tutorial for *simplenlg (version 3.8)*

by Chris Venour, Ehud Reiter and Dave Westwater

28 September 2009

Table of Contents

Section I - What is simplenlg?....3
Section II –Who uses simplenlg?....4
Section VI - Verbs....10
Simplenlg can also generate simple WH questions. For example....11
Section VII – What are complements?....12
Section VIII -Adding multiple subjects and complements....13
Section IX – Adding adjectives via 'modifier'....14
Section X – Prepositional phrases....15
Section XI – Adding adverbs 18
Section XIII – Different ways of specifying a phrase....20
Section XIV- Generating a sentence with multiple clauses....21
Section XV - Generating paragraphs (TextSpec)....23
Section XVI - Generating a paragraph with strings, SPhraseSpecs and TextSpecs....24
Appendix A – NLG and simplenlg....25
Appendix B – simplenlg web pages....28
Appendix C – simplenlg and Eclipse....29

Section I - 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.

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, for example, to specify the subject of a sentence ('my dog'), the exact verb you want to appear in the sentence ('chase') and the object ('George'). You might also make use of *simplenlg* methods to indicate that you would like the verb to be in the past tense and expressed in the progressive form.

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". Here, *simplenlg* has

- capitalized the first letter of the sentence
- added the auxiliary 'was' and made it agree with the subject
- added '-ing' to the end of the verb (because the progressive aspect of the verb is desired)
- put all the words together in a grammatical form
- inserted the appropriate whitespace between the words of the sentence
- 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 almost entirely the exact 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. 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. (Inflection is the modification or marking of 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[2]. 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.

Section II –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 realization.

- Students who want to learn more about natural language generation.

- Anyone who wants to write programs that can generate English sentences.

---

[1] **agreement** describes how a word's form sometimes depends on other words that appear with it in a sentence. For example you don't say "I is" in English, because "is" cannot be used when the subject is "I". The word "is" is said not to agree with the word "I". The correct form is "I am", even though the verb still has the same function and basic meaning.

[2] What is this simple grammar? See Appendix B for details.

Section III - Getting started[3]

It's important to note that *simplenlg* is a library[4], 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 a zip file of the library from the *simplenlg* website (see Appendix B for the URL).
- unload that zip file and add *simplenlg*'s jar file to your program's build path (For instructions on how to do this in Eclipse, see Appendix C).
- 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.features.*;
import simplenlg.realiser.*;
```

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

[3] Note that this and other sections assume a basic understanding of object oriented programming on the part of the reader. For example, you will need to understand what a class and an instance of a class are.
[4] 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.

```
import simplenlg.features.*;
import simplenlg.realiser.*;


public class TestMain {


   public static void main(String[] args) {


   }

}
```

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!

The main steps involved in generating a sentence:

Importing the *simplenlg* library into your Java program provides that program with classes that allow you to specify the parts of speech of a sentence. The idea of 'what a sentence is' is captured in the *simplenlg* class called **SPhraseSpec**; the idea of a 'noun phrase' is represented by the class **NPPhraseSpec** and the notion of 'prepositional phrase' is represented by the class **PPPhraseSpec**[5]. In order to build a sentence with these *simplenlg* concepts or classes, you will normally follow these steps:

- create an instance of **SPhraseSpec**. (This represents our sentence).
- create instances of various other parts of speech (using **NPPhraseSpec** or **PPPhraseSpec** for instance).
- indicate what role these various parts of speech will play in the desired **SPhraseSpec** sentence. For example, specify that you want a particular noun phrase to be the subject of the **SPhraseSpec** sentence, and some other noun phrase to be the object.
- specify what the verb of **SPhraseSpec** will be.
- create a *simplenlg* object called the **Realiser**.
- ask the **Realiser** to 'realise' or transform the **SPhraseSpec** instance into a syntactically correct string.

[5] *simplenlg* has no concept of two other important phrase types: adjective phrases and adverb phrases. These parts of speech are generated in another way, using the *simplenlg* concept of **modifier**. See section IX for details.

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**.

It's important to note that you only need to create the Realiser once. i.e.you don't need to create it for every sentence you generate. So a good idea is to create a single realiser at the start of your program and feed it various sentence specifications over the lifetime of the program run.

See **Section V** for an example of the actual Java code used to generate a sentence.

Section IV – Generating words (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 used by default. It can also be accessed explicitly to find out different forms of words, as below

```
import simplenlg.lexicon.Lexicon;
Lexicon lex = new Lexicon();

System.out.println(lex.getPlural("child"));
System.out.println(lex.getPast("eat"));
System.out.println(lex.getPastParticiple("eat"));
System.out.println(lex.getPresent3SG("eat"));
System.out.println(lex.getPresentParticiple("eat"));
System.out.println(lex.getComparative("happy"));
System.out.println(lex.getSuperlative("happy"));
```

These statements will print out the following words:

```
children
ate
eaten
eats
eating
happier
happiest
```

Simplenlg's built-in lexicon just contains information about word forms. Simplenlg can also access a lexicon that is held in an external database, which can contain much more information about words. In particular, simplenlg can be used with a modified version of the specialist lexicon developed for the National Institute of Health in the USA (see http://specialist.nlm.nih.gov/ for more information about the specialist lexicon).

We are still developing this aspect of simplenlg. To use this lexicon, you must first load it into a local database, such as mysql (http://www.mysql.com/); we provide a mysql version of the lexicon on the simplenlg web page as lexicon.sql. Please contact your system administrator if you are not familiar with mysql.

When the database has been loaded, simplenlg can access by creating a lexicon class as follows

```
Lexicon lex = new DBLexicon("com.mysql.jdbc.Driver",
                            "jdbc:mysql://localhost/lexicon",
                            "lexicon", "password");
```

The parameters to the DBLexicon call are the standard ones used to set up JDBC databases in Java: the first is the driver, the second is the database location, the third is the database username, and the fourth is the database password. Again ask your system administrator if you are not familiar with JDBC.

When the lexicon has been set up, you can retrieve information about particular words using the getItem() method, and then get information about particular aspects of this word. For example,

```
Noun mouse = (Noun) lex.getItem(Category.NOUN, "mouse");
System.out.println(mouse.getplural());
System.out.println(mouse.isCountNoun());
```

This will print out

```
mice
true
```

The simplenlg lexicons do not contain information about semantic relations of a word. For instance you can't ask *simplenlg* to output a synonym of the word 'happy' because it does not possess a lexicon with that kind of information in it.

Section V – Generating a simple sentence

In the Java class `TestMain` shown above, we have the statement

```
import simplenlg.realiser.*;
import simplenlg.features.*;
```

These classes allow you to specify the parts of speech of a sentence and to perform various operations on them. The idea of 'what a sentence or phrase is' is captured in the *simplenlg* class called **SPhraseSpec**; the idea of a 'noun phrase' is represented by the class **NPPhraseSpec** and the notion of 'prepositional phrase' is represented by the class **PPPhraseSpec**. There are no classes defined for adjective phrases or adverb phrases in *simplenlg*: these parts of speech are generated using something called a **modifier** which is described in Section IX.

It's important to note that *simplenlg* provides only a very simple grammar: its notions of a sentence, noun phrase and prepositional phrase are very basic and are by no means representative of the incredibly varied and complicated grammar of the English language[6].

Let's see how we would define and combine various parts of speech to generate a simple sentence such as "My dog chases George". 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. Dividing a sentence into smaller parts and labeling those parts, provides more flexibility because we can then reassemble components in various ways.

```
SPhraseSpec p = new SPhraseSpec();
p.setSubject("my dog");
p.setVerb("chase");
p.addComplement("George");
```

The above set of calls to *simplenlg* defines the constituents or components of the sentence we wish to construct: we have specified a subject, a verb and an object/complement (*simplenlg* uses the term complement) for our sentence. Now, all that remains is to create a 'realiser' which will take these different components of the sentence, combine them and 'realise' the text to make the result syntactically and morphologically correct:

```
Realiser r = new Realiser();
String output = r.realiseDocument(p);
System.out.println(output);
```

The resulting output is:

[6] See **Appendix B** for a full description of the simplenlg grammar.

```
My dog chases George.
```

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 **addComplement**, assemble the parts of speech by obeying the simple grammar embodied in *simplenlg*[7].

Section VI - Verbs

Verbs should be specified in infinitive ("to XXX") form. However, as a convenience, *simplenlg* will recognize inflected forms of "be" such as "am". 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 George":

SPhraseSpec p = new SPhraseSpec();
p.setSubject("Mary");
p.setVerb("chase");
p.addComplement("George");

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

p.setTense(Tense.PAST);

thus rendering the sentence to:

```
Mary chased George.
```

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

p.setTense(Tense.FUTURE);

yielding the sentence:

[7] 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.

```
Mary will chase George.
```

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.setNegated(true);

will change the resulting sentence to

```
Mary will not chase George.
```

Questions

Simplenlg can generate simple yes/no questions. For example

p.setSubject("Mary");
p.setVerb("chase");
p.addComplement("George");
p.setInterrogative(InterrogativeType.*YES_NO*);

will generate

```
Does Mary chase George?
```

Simplenlg can also generate simple WH questions. For example

p.setSubject("Mary");
p.setVerb("chase");
p.setInterrogative(InterrogativeType.*WHO*, DiscourseFunction.*OBJECT*);

will generate

```
Who does Mary chase?
```

Section VII – What are complements?

In the previous example, the object of the verb is "George" and it's called the complement. So what is a complement exactly? As far as *simplenlg* is concerned, a complement is anything that comes immediately 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[8].

Examples of complements are underlined in the sentences below:

1. John ate an apple.
2. John is happy.
3. John wrote quickly.
4. John just realized that his holidays are over.

The underlined words and phrases in the examples above are all different parts of speech. In example #1, the complement is a noun phrase; in example #2, it's an adjective; in example #3 an adverb and it's a 'that-clause' in example #4. But from *simplenlg*'s point of view, the underlined bits are none of these things: they are simply complements because they all appear after the verb. Although it has a (very basic) understanding of verbs, noun phrases and prepositional phrases, *simplenlg* has no concept of adjectives, adverbs, 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, parts of speech which appear after a verb can be generated using the *simplenlg* library.

Phrase Type	**Examples**
Noun Phrase	"an apple"
Prepositional Phrase	"in the park"
Verb	"chase"
Verb Phrase	"play the piano"
Adjective Phrase	"delighted to meet you"
Adverb Phrase	"very quickly"

Table 1: Highlighted are the parts of speech that *simplenlg* can explicitly handle. The concepts of adjective phrases and adverb phrases, however, fall under the headers 'complement' or 'modifier'.

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

Note that only things of type **SPhraseSpec** can take a complement.

Section VIII -Adding multiple subjects and complements

An **SPhraseSpec** can have multiple subjects and complements but not multiple verbs (although a future version of *simplenlg* might include this functionality). Let's say you have a monkey that also wants to chase poor George. To add your monkey to the fray, you would write:

```
p.addSubject("your monkey");
```

The resulting output is:

```
Mary and your monkey chase George.
```

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 complements in an **SPhraseSpec.** Let's suppose Mary and the monkey have found more people to terrorize in what's turning out to be a growing parade of horror:

```
p.addComplement("Joey");
p.addComplement("Martha");
```

The resulting output will be:

```
Mary and your monkey chase George, Joey and Martha.
```

If you wish to combine subjects or complements with "or" instead of "and", you can do this by creating a CoordinateNPPhraseSpec; this is described in the API.

Section IX – Adding adjectives via 'modifier'

We know for a fact that George et al. don't like to be chased, at least not on a full stomach, so we'd like to assign Mary and the monkey a suitable adjective. The problem is that, although *simplenlg* knows what noun phrases, verbs and prepositional phrases are, it has no concept of what an adjective (or adverb are). But don't worry! These parts of speech can be still be generated by the *simplenlg* library – it just doesn't label them as such. Instead they are subsumed under the larger concept of **modifier**.

To deem Mary and the monkey 'cruel', however, you will no longer want to refer to them simply as 'subjects' of the sentence. Instead let's also define them as noun phrases (which they are). In that way we can ascribe the adjective 'cruel' (which they certainly are) to those noun phrases by means of the **modifier** function.

```
NPPhraseSpec subject1 = new NPPhraseSpec("Mary");
NPPhraseSpec subject2 = new NPPhraseSpec("your", "monkey")[9];
```

Now, we can apply the adjective 'cruel' to these noun phrases by writing:

```
subject1.addModifier("cruel");
subject2.addModifier("cruel");
```

With the rest of the code in place (assuming that a Realiser r has been created already):

```
p.setSubject(subject1);
p.addSubject(subject2);
p.setVerb("chase");
p.addComplement("George");
p.addComplement("Joey");
p.addComplement("Martha");

String output = r.realiseDocument(p);
System.out.println(output);
```

The output will be:

```
Cruel Mary and your cruel monkey chase George, Joey
and Martha.
```

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

```
NPPhraseSpec subject2 = new NPPhraseSpec("monkey");
subject2.setDeterminer("your");
```

Section X – Prepositional phrases

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

"Mary chases George **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 most simplistic way would be to simply label the string "in the park" as a **PPPhraseSpec** and add it as a **modifier** of the sentence:

```
SPhraseSpec p = new SPhraseSpec("Mary", "chase", "George");
PPPhraseSpec pp = new PPPhraseSpec("in", "the park");
p.addModifier(pp);
```

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

We then add the prepositional phrase as a modifier of the 'Mary chases George' sentence.

```
p.addModifier(pp);
```

The table below shows these two different ways of creating the prepositional phrase "in the park".

more simplistic way of adding a prepositional phrase	more 'sophisticated' way
SPhraseSpec p = new SPhraseSpec("Mary", "chase", "George"); **PPPhraseSpec pp = new PPPhraseSpec("in", "the park");** **p.addModifier(pp);**	SPhraseSpec p = new SPhraseSpec("Mary", "chase", "George"); **NPPhraseSpec place = new NPPhraseSpec("park");** **place.setDeterminer("the");** **PPPhraseSpec pp = new PPPhraseSpec();** **pp.addComplement(place);** **pp.setPreposition("in");** **p.addModifier(pp);**
Mary chases George in the park	

Table 2: Two ways of adding the prepositional phrase 'in the park' to a sentence.

The more simplistic way requires less code than the second 'more sophisticated' way. So why then would we ever choose the second method?

The main reason is that the second 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 describe the park as 'leafy' and we had used the 2nd method to define our sentence, all we would need to do is write the following code:

place.addModifier("leafy");

Had we chosen the first method, however, adding the adjective 'leafy' to the string 'in the 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 easy 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 method #1? 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 monolithic entity the way method #1 does.

Section XI – Adding adverbs

Adverbs are specified as modifiers in an SPhraseSpec. For example, to output "John quickly ran", use

```
p.setSubject("John");
p.setVerb("run");
p.setPremodifier("quickly");
p.setTense(Tense.PAST);
```

Section XII – Modifiers vs complements

There are four 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**) and adjective phrases (which aren't discussed here). Note that other parts of speech such as adverb phrases are not explicitly handled by *simplenlg* but they can certainly be generated using the *simplenlg* concepts of modifier and complement.

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

Section XIII – Different ways of specifying a phrase

There are numerous ways of specifying a phrase. The table below shows some of the ways we can create the sentence 'Mary chases George'. 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).

1.	**SPhraseSpec p = new SPhraseSpec("Mary", "chase", "George");**
2.	**SPhraseSpec p = new SPhraseSpec();** **p.addSubject("Mary");** **p.setVerb("chase");** **p.addComplement("George");**
3.	**NPPhraseSpec subj = new NPPhraseSpec("Mary");** **NPPhraseSpec obj = new NPPhraseSpec("George");** **SPhraseSpec p = new SPhraseSpec(subj, "chase", obj);**
4.	**NPPhraseSpec obj = new NPPhraseSpec("George");** **SPhraseSpec p = new SPhraseSpec();** **p.addSubject("Mary");** **p.setVerb("chase");** **p.addComplement(obj);**
5.	**SPhraseSpec p = new SPhraseSpec("Mary", "chase", new NPPhraseSpec("George"));**

Table 4: different ways of creating the sentence "Mary chases George".

Section XIV- Generating a sentence with multiple clauses

You can generate a sentence with multiple clauses in two ways:
1. by using a *simplenlg* class called **TextSpec**
2. by nesting phrases within phrases

Lists of clauses

One way of generating a sentence with multiple clauses is to use the *simplenlg* class **TextSpec**. **TextSpec** can be used to define single sentences and paragraphs. It consists of a document structure (eg SENTENCE or PARAGRAPH) and a list of components which are either **SPhraseSpecs** or smaller **TextSpecs**.

In the next section we will see how to create paragraphs with **TextSpec** but for now let's see how we can create a list of clauses which we want combined in a single sentence:

```
TextSpec t1 = new TextSpec("my cat likes fish", "my dog likes bones", "my
horse likes grass");
```

We can define the **TextSpec** instance t1 to be made up of a list of **SPhraseSpecs** in this way or we can make use of the method **addSpec** and individually add each of them to t1:

```
SPhraseSpec s1 = new SPhraseSpec("my cat", "like", "fish");
SPhraseSpec s2 = new SPhraseSpec("my dog", "like", "bones");
SPhraseSpec s3 = new SPhraseSpec("my horse", "like", "grass");

TextSpec t1 = new TextSpec();  // create a TextSpec
t1.addSpec(s1);
t1.addSpec(s2);
t1.addSpec(s3);
```

If you do not supply a conjunction using the method **setListConjunct**, 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 bones and my horse
likes grass.
```

If you do not want a conjunction to appear in the sentence, specify the empty string "" as the list conjunct. For the t1 list specified above, you would write:

```
t1.setListConjunct("");
```

The resulting output would be[10]:

```
My cat likes fish, my dog likes bones my horse likes
```

[10] bug here: a final comma is missing

```
grass.
```

Subordinate clauses:
You can use the same **TextSpec** methods as the ones mentioned above to create a sentence that has a main and subordinate clause. Note that the order in which you add phrases to the **TextSpec** matters: that order indicates their left to right order in the resulting text.

```
SPhraseSpec s1 = new SPhraseSpec("I", "be", "happy");
SPhraseSpec s2 = new SPhraseSpec("I", "eat", "fish");
s2.setCuePhrase("because");
s2.setTense(Tense.PAST);

TextSpec t1 = new TextSpec(); // create a TextSpec
t1.addSpec(s1);
t1.addSpec(s2);

String output = r.realiseDocument(t1);  //Realiser r created earlier
System.out.println(output);
```

The output is:

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

Section XV - Generating paragraphs (TextSpec)

As we saw in the previous section, **TextSpec** allows you to create a sentence consisting of a list of clauses. For example,

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

The code for producing this output is

TextSpec t1 = new TextSpec("my cat likes fish", "my dog likes bones", "my horse likes grass");

TextSpecs are sentences by default but they can also be defined to generate paragraphs. To create a paragraph you need to:

1. create an instance of **TextSpec**
2. add sentences to that instance (using the method **addSpec**) or include the sentences in the constructor[11] for **TextSpec**.
3. define the **TextSpec** instance as a paragraph rather than a single sentence (using the method **setParagraph)**.

Thus if we add the statement:

t1.setParagraph();

to the code above, the realiser will produce the following paragraph:

```
My cat likes fish. My dog likes bones. My horse
likes grass.
```

If you do not set the **TextSpec** instance to be a paragraph, *simplenlg* will, by default, treat it as a single sentence.

[11] 'constructor' is a Java term. It's the line of Java code in which an instance of a class is created. The constructor I'm referring to here is: **TextSpec t1 = new TextSpec("my cat likes fish", "my dog likes bones", "my horse likes grass");**

Section XVI - Generating a paragraph with strings, SPhraseSpecs and TextSpecs

You can combine strings, **SPhraseSpecs** and **TextSpecs** in a single **TextSpec**. See the example below[12].

```
String str1 = "John is going to Tesco";
String str2 = "Mary is going to Sainsburys";
TextSpec t1 = new TextSpec();  // create a TextSpec
t1.addSpec(str1);
t1.addSpec(str2);

SPhraseSpec p1 = new SPhraseSpec("I", "go", "school");
p1.setCuePhrase("however");
p1.setProgressive(true);

TextSpec t2 = new TextSpec();
t2.addSpec(t1);
t2.addSpec(p1);

String output = r.realiseDocument(t2);  //Realiser r created earlier
System.out.println(output);
```

Output: `John is going to Tesco and Mary is going to Sainsburys. However I am going to school.`

In this example, we used the cue phrase 'however' to smooth the transition from the first sentence to the next one. Cue phrases often express how a sentence relates to the previous clause or sentence and help sentences in a paragraph flow together.

Notice that in this example we did not need to indicate that we want to generate a paragraph – *simplenlg* automatically did this for us i.e. we did not have to include the statement **t2.setParagraph**. Why not? Because if you add a TextSpec (t1 in this example) to another TextSpec (t2), they will automatically be considered separate sentences.

[12] Note that *simplenlg* doesn't always add all the punctuation that it should: there should be a comma after "However".

Appendix A – NLG and *simplenlg*

What is NLG and how much of it does *simplenlg* do? NLG aims to produce understandable text, typically from some nonlinguistic representation of info.

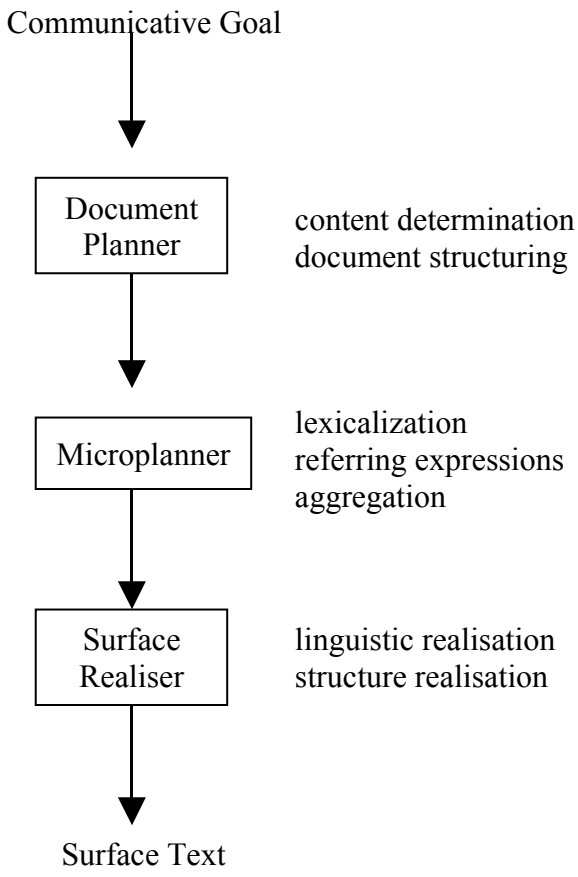

Figure 2: A typical NLG system architecture. From Reiter and Dale p.60.

Many NLG systems consist of 3 components which are connected together in a pipeline. i.e. the output of document planning acts as input to microplanning and the output of the microplanner is the input to the surface realiser. The table below briefly outlines the different components of an NLG system and the shaded portion shows which tasks *simplenlg* performs.

Document Planner	**content determination**	decides what information will appear in the output text. This depends on what your goal is, who the audience is, what sort of input information is available to you in the first place and other constraints such as allowed text length.
	document structuring	decides how chunks of content should be grouped in a document, how to relate these groups to each other and in what order they should appear. For instance, when describing last month's weather, you might talk first about temperature, then rainfall. Or you might start off generally talking about the weather and then provide specific weather events that occurred during the month.
Microplanner	**lexicalization**	decides what specific words should be used to express the content. For example, the actual nouns, verbs, adjectives and adverbs to appear in the text are chosen from a lexicon. Particular syntactic structures are chosen as well. For example you can say 'the car owned by Mary' or you might prefer the phrase 'Mary's car'.
	referring expressions	decides which expressions should be used to refer to entities (both concrete and abstract). The same entity can be referred to in many ways. For example March of last year can be referred to as: • *March 2006* • *March* • *March of the previous year* • *it*
	aggregation	decides how the structures created by document planning should be mapped onto linguistic structures such as sentences and paragraphs. For instance, two ideas can be expressed in two sentences or in one: *The month was cooler than average.The month was drier than average.* vs. *The month was cooler and drier than average.*

Surface Realiser	**linguistic realisation**	uses rules of grammar (about morphology and syntax) to convert abstract representations of sentences into actual text.
	structure realization	converts abstract structures such as paragraphs and sentences into mark-up symbols which are used to display the text.

Table 5: the shaded portion of the table shows how much nlg *simplenlg* performs

Appendix B – *simplenlg* web pages

To download *simplenlg*, go to http://www.csd.abdn.ac.uk/~ereiter/simplenlg/

The web page also contains detailed API documentation for *simplenlg*

Appendix C – *simplenlg* and Eclipse

Setting up *simplenlg* in Eclipse:
If you are using Eclipse to write your Java programs, you would

1. create a new Project (File>New>Project)
2. add the *simplenlg* library to the project's build path by doing the following:

- go to the Eclipse menu and select Project>Properties>Java Build Path
- click on the Libraries tab
- click on the Add External jars button
- browse to the *simplenlg* jar file which you downloaded from the web
- select the *simplenlg* jar file and click 'Open'
- click OK

3. In the project, create a new class which has the main method in it. In this example, let's call the new class `TestMain`.

4. At the top of the class, put in the following import statements:

```
import simplenlg.features.*;
import simplenlg.realiser.*;
```

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

```
import simplenlg.features.*;
import simplenlg.realiser.*;

public class TestMain {

    /**
     * @param args
     */
    public static void main(String[] args) {
          // TODO Auto-generated method stub

    }

}
```

Figure 3: a Java class which is ready to make use of the *simplenlg* library