Overview of Simplenlg Package (May 2008)(v3.7)

Simplenlg is a relatively simple Natural Language Generation realiser, with a Java API. It has less grammatical coverage than many other realisers, but it does not require in-depth knowledge of a syntactic theory to use. The core of the package is the realiser, lexicon, and features packages; the KB package is more experimental.

This is largely based on the architecture described in Reiter and Dale (2000). It is very much under development, comments welcome!

This document gives an overview of the most important classes and methods. Please see the API documentation for full details. If you are a new user, you may wish to read our tutorial first.

Conditions of Use 2
Overview of Main Simplenlg Classes 3
Realiser class 4
Lexicon class 5
  DBLexicon class 6
Spec class (abstract) 7
  PhraseSpec class (abstract) 7
  StringPhraseSpec class 7
  HeadedPhraseSpec class (abstract) 7
  SPhraseSpec class 8
  NPPhraseSpec class 9
  PPPhraseSpec class 10
  AdjPhraseSpec class 11
  CoordinatePhraseSpec interface 12
TextSpec class 13
NLGKB interface 14
  KBEntity interface 14
  JavaKB class 15
  ProtegeKB class 16
  SimpleKB class 17
  SimpleKBClass class 17
Lexicaliser class 18
Design Rationale 20
References 21

Conditions of Use

This library contains a re-implementation of some of the rules used in the MorphG package (Minnen, Carroll and Pearce 2001). Thanks to John Carroll for permission to re-use the MorphG rules.

Overview of Main Simplenlg Classes

Simplenlg.realiser (Main classes)

- Spec – specifies a phrase
  - PhraseSpec – specifies a phrase, such as “John is happy”
    - StringPhraseSpec – specifies a PhraseSpec as a string
    - HeadedPhraseSpec – specifies a PhraseSpec as a syntactic structure
      - SPhraseSpec – specifies a sentence
      - NPPhraseSpec – specifies a noun phrase
      - PPPhraseSpec – specifies a prepositional phrase
      - AdjPhraseSpec – specifies an adjective phrase
      - CoordinatePhraseSpec – specifies coordinated phrase spec
  - TextSpec – specifies a collection of PhraseSpecs (or smaller TextSpec’s)
- Realiser – generates a text from a Spec
- DocumentStructure – enumerated type, specifies sentence/paragraph/etc

Simplenlg.kb (Main classes)

- NLGKB – interface to a knowledge base (only needed for microplanning)
  - JavaKB – KB represented as Java classes
  - ProtegeKB – KB represented in Protégé
  - SimpleKB – explicitly constructed KB
- KBEntity – interface to entity (class or instance) in an NLGKB
- Lexicaliser – converts KBEntity to PhraseSpec (different from Lexicon!)

Simplenlg.lexicon (Main classes)

- Lexicon – computes word inflections (eg, plurals)
- DBLexicon – accesses lexical information in a database

Simplenlg.features (Main classes)

- Form – form of a clause (Normal, Imperative, etc)
- Mood – mood of a clause (Normal, Subjunctive)
- Tense – tense of a clause (Past, Present, Future)

Simplenlg.exception (Main classes)

- SimplenlgException – exception detected by simplenlg package

Simple Usage Example

```
PhraseSpec s1 = new StringPhraseSpec("my dog is happy");        // first sentence
PhraseSpec s2 = new SPhraseSpec("my cat", "is", "sad");         // second sentence

TextSpec t = new TextSpec(s1,s2);          // text spec for document as a whole

Realiser r = new Realiser();               // set up realiser
String output = r.realiseDocument(t);      // generate text from TextSpec
System.out.println(output);                // print out the text
```

This will print out

```
My dog is happy and my cat is sad.
```

Realiser class

A realiser converts a TextSpec into text. The text can be formatted as simple text or as HTML (MS Word RTF may be added at some point). Newlines are added so that the text fits into 70-character lines (by default).

Constructors

```
Realiser();                 // returns text realiser
Realiser(Lexicon lex);      // realiser using specified lexicon
```

Key methods

```
void setHTML(html);         // if html is true, include HTML markups
void setLineLength(length); // change the line length (default is 70)
String realise(Object spec);    // realise a specification
String realiseDocument(Object spec); // realise a spec as a document
```

Realisers automatically take care of low-level processing such as adding spaces between phrases when necessary, adding line breaks if lines get too long, forming plural forms of verbs if necessary, etc.

Example

```
t = new TextSpec("This is a test of the simplenlg package developed by Ehud Reiter");

Realiser r = new Realiser();                // set up realiser
r.setHTML(true);
r.setLineLength(40);
String output = r.realiseDocument(t);       // generate text from TextSpec
System.out.println(output);                 // print out the text
```

This will print out

```
<P>This is a test of the simplenlg package
developed by Ehud Reiter.
```

Lexicon class

This holds information about irregular forms of words. Simplenlg's Lexicon is partially based on the Sussex *morphg* system (Minnen, Carroll, Pearce 2001). See the Lexicon API for full details about this class. Note that Lexicon is in the "simplenlg.lexicon" package, not the "simplenlg.realiser" package.

Constructors:

```
Lexicon();                          // returns a default lexicon
```

Key methods

```
String getPlural(String noun);          // return plural form of a noun
String getPast(String verb);            // return past tense of a verb
String getPastParticiple(String verb);  // return past participle of a verb
String getPresent3SG(String verb);    // return present third sing of verb
String getPresentParticiple(String verb);  // return present participle of verb
String getComparative(String adj);    // get comparative of adjective
String getSuperlative(String adj);      // get superlative of adjective
```

The adjective methods are experimental, and may not always work

See the full Lexicon documentation for details of how to add new word to a lexicon (this should not usually be necessary, as the coverage of the default lexicon is quite good).

Example

```
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.getPresent3S("eat"));
System.out.println(lex.getPresentParticiple("eat"));
System.out.println(lex.getComparative("happy"));
System.out.println(lex.getSuperlative("happy"));
```

This will print out

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

DBLexicon class

A DBLexicon accesses lexical information held in an external database. Currently simplenlg is set up to use a modified version of the NIH Specialist lexicon (http://specialist.nlm.nih.gov/). This provides much more information about words than the simple morphology present in the default lexicon. At the moment this information is not used by simplenlg's realiser, its just a resource which developers can access; in the future however the realiser may be guided by the lexical information. The lexicon returns classes defined in simplenlg.lexicon.lexicalitems. See the API for detailed information

Constructors:

```
DBLexicon(String driver, String URL, String username, String password);
                // creates a lexicon based on the specified DB
                // parameters are standard JDBC DB specification
```

Key methods

```
LexicalItem getItem(Category, String base);  // return lexical item
LexicalItem getItemByID(String ID);  // return item with specified ID inDB
void loadData();                     // load all items in DB
```

LexicalItems have methods such as getPlural(), isCountNoun(), isIntransitive(), etc. Again see API for details. Please note that if you load the entire Specialist lexicon, you will need to make sure Java has enough memory; we suggest specifying -Xmx1g (1GB memory) (or more) as a run-time Java flag.

Example

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

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

This will print out

```
mouse
true
```

Spec class (abstract)

A Spec is either a PhraseSpec or a TextSpec

PhraseSpec class (abstract)

A PhraseSpec defines a phrase. Note that the realiser will apply sentence orthography (capitalise first letter, add full stop) if appropriate, so you don't need to do this; in fact its best if you don't do this, in case the realiser wants to do things like add a cue phrase at the beginning of a sentence, or merge two phrases into a single sentence. For example

```
PhraseSpec s1 = new StringPhraseSpec("my dog is happy");
```

is preferred over

```
PhraseSpec s1 = new StringPhraseSpec("My dog is happy.");
```

StringPhraseSpec class

A StringPhaseSpec defines a phrase as a String. It has the obvious constructor

```
StringPhraseSpec(String phrase)
```

In the terminology of Reiter and Dale (2000) (page 68), a StringPhaseSpec is canned text, not orthographic string.

HeadedPhraseSpec class (abstract)

A HeaderPhraseSpec defines a phrase via its syntactic constituents; it must have a head. It is similar to Reiter and Dale's Abstract Syntactic Structure (page 69)

SPhraseSpec class

A SPhraseSpec defines a sentence or clause in terms of the following constituents

verb - eg, “eat”, “run”, “be”
subjects - eg, “my dog”, “the cat” (ie, what is mentioned before the verb)
complements – eg, “an apple”, “happy” (ie, what is mentioned after the verb)
indirectObjects – eg, "the book" in "John gave Mary the book"
post modifiers – eg, “in the park” (which occur at the end of a sentence)
pre modifiers – eg, "quickly" (which occur just before the verb)
front modifiers – eg, “unfortunately” (which occur at the start of a sent)
cue phrase – eg, “However” (relationship with previous clause)

Many features can also be specified

tense – Tense.PAST, Tense.PRESENT, Tense.FUTURE (enum type)
form – Form.NORMAL, Form.IMPERATIVE, Form.INFINITIVE, Form.GERUND
mood – Mood.NORMAL, Mood.SUBJUNCTIVE
interrogative – InterrogativeType.YES_NO, InterrogativeType.WHO, etc
negated – Boolean, specifies if phrase is negated
progressive – Boolean, specifies progressive aspect if True
passive – Boolean, specifies passive voice if True
perfect – Boolean, specifies perfect aspect if True
modal – String, specifies modal auxiliary (eg, "must")

Please see the API for constructors and methods, there are too many to list here

Example:

```
SPhraseSpec p = new SPhraseSpec();
p.addSubject("my dog");
p.addSubject("your cat");
p.setVerb("chase");
p.addComplement("George");
p.setTense(Tense.PAST);
p.setProgressive(true);
p.setNegated(true);
p.setPassive(true);
System.out.println(r.realiseDocument(p));
```

This will print out

```
Yesterday, George was chased by my dog and your cat.
```

Please see the tutorial for simple example of how to use SPhraseSpec

NPPhraseSpec class

A NPPhraseSpec defines a noun phrase in terms of the following constituents
- determiner – eg, “the”, “your”, “a”
- preModifiers – eg, “happy”, “red” (adjs and other mods before the noun)
- noun – eg, “dog”, “cat”
- postModifiers – eg, “in the park” (prep phrases and other mods after the noun)

Some features can also be specified
- plural – make NP plural (eg, “dogs” instead of “dog”)
- possessive – return possessive form (eg, `dog's` instead of `dog`)

NPPhraseSpec can be specified as subjects and complements in an SPhraseSpec, and as object in a PPPhraseSpec.

simplenlg automatically recognises most pronouns (eg, “I”, “us”, “him”), and does some very simple processing of determiners (eg, “a” vs “an”).

Key constructors

```
NPPhraseSpec();
NPPhraseSpec(String noun) ;
NPPhraseSpec(String determiner, String noun);
```

Key methods

```
void addModifier(Object modifier);
void addPremodifier(Object modifier);
void addPostmodifier(Object modifier);
void setDeterminer(String determiner);
void setNoun(String noun);

void setPlural(boolean elided);
void setPossessive(boolean possessive);
```

Example

```
NPPhraseSpec subject = new NPPhraseSpec(“He”);
NPPhraseSpec object = new NPPhraseSpec(“the”, “dog”);
object.setPlural(true);
object.addModifier(“big”);
object.addModifier(“red”);
SPhraseSpec p = new SPhraseSpec(subject, “hate”, “object”);
```

results in the text

```
He hates the big red dogs.
```

PPPhraseSpec class

A PPPhraseSpec defines a prepositional phrase in terms of
preposition – eg, “in”, “after”
object – eg, “the park”, “5PM”

A PPPhraseSpec can have multiple objects.

PPPhraseSpec’s do not have features

Key constructors
PPPhraseSpec();
PPPhraseSpec(String preposition);
PPPhraseSpec(String preposition, Object object);

Key methods
void addObject(Object object);
void setPreposition(String preposition);

Example
PPPhraseSpec pp = new PPPhraseSpec("in", new NPPhraseSpec("the", "park"));
p = new SPhraseSpec("I", "be");
p.addModifier(pp);

results in the text
```
I am in the park.
```

AdjPhraseSpec class

An AdjPhraseSpec defines a adjective phrase in terms of
adjective – eg, “big”, “red”
preModifier – eg, “very"

Key constructors
AdjPhraseSpec();
AdjPhraseSpec(String adjective);

Key methods
void setHead(Object object); // used to set the adjective
void addPremodifier(String modifier);

Example
```
AdjPhraseSpec adj = new AdjPhraseSpec("big");
adj.addPremodifier("very");
NPPhraseSpec np = new NPPhraseSpec("a", "dog");
np.addPremodifier(adj);
SPhraseSpec s = new SPhraseSpec("I", "see", np);
```

results in the text
```
I see a very big dog.
```

CoordinatePhraseSpec interface

Simplenlg includes several classes for coordinate structures, including CoordinateAdjPhraseSpec, CoordinateNPPhraseSpec, CoordinatePPPhraseSpec, and CoordinateSPhraseSpec. These all implement the CoordinatePhraseSpec interface.

Coordination means having several NPs (or whatever) joined together, eg "John and Mary" in "John and Mary see the dog". In fact, simplenlg can handle this case without using an explicit CoordinatePhraseSpec;

```
SPhraseSpec s = new SPhraseSpec();
s.addSubject("John");
s.addSubject("Mary");
s.setVerb("see");
s.setComplement("the dog");
```

If this default behaviour is not acceptable, a CoordinatePhraseSpec can be used. This for example allows a different conjunction to be used (eg, "John or Mary" instead of "John and Mary") and also allows control of scoping (eg, "the man and the woman" vs "the man and woman").

Key constructors

```
CoordinateNPPhraseSpec(NPPhraseSpec …);
CoordinateAdjPhraseSpec(AdjPhraseSpec …);
etc
```

Key methods

```
void addCoordinates(PhraseSpec …) ;
void setConjunction(String conjunct)
```

Example

```
NPPhraseSpec n1 = new NPPhraseSpec("the", "apple");
NPPhraseSpec n2 = new NPPhraseSpec("the", "pear");
CoordinateNPPhraseSpec n3 = new CoordinateNPPhraseSpec(n1,n2);
n3.setConjunction("or");
```

results in the text

```
the apple or the pear
```

TextSpec class

A TextSpec defines a sentence, paragraph, or other higher-level document structure. It essentially consists of a document structure (eg, SENTENCE) and a list of components (which are PhraseSpecs or smaller TextSpecs). In other words, TextSpecs form a tree (as discussed in Reiter and Dale)

Key constructors are

TextSpec();
TextSpec(Object … spec); // any number of specs allowed. Can be String,
TextSpec (DocStructure, Object … Spec); // PhraseSpec, TextSpec

Key methods are

void addSpec(Object spec); // add another child spec
void setDocStructure(DocStructure docStructure); // set any doc structure
void setSentence(); // set doc structure to sentence (this is default)
void setParagraph(); // set doc structure to paragraph
void setDocument(); // set doc structure to document
void setListConjunct(String); // set conjunct, used to combine components

Allowable document structures are defined in the DocStructure enum class. They are PHRASE, PHRASESET, SENTENCE, SENTENCESET, PARAGRAPH, PARAGRAPHSET, DOCUMENT. TextSpecs are Sentences by default.

If a list conjunct is specified, it will be used to conjoin components (see below). If it isn't specified, "and" is the default conjunct for sentential components that do not have cue phrases (if you do not want a conjunct, specify "" as the list conjunct). See example below.

Example

```
TextSpec t1 = new TextSpec("my cat likes fish", "my dog likes bones", "my horse likes grass");
TextSpec t2 = new TextSpec("John is going to Tesco", "Mary is going to Sainsburys");
t2.setListConjunct("or");
SPhraseSpec p1 = new SPhraseSpec("i", "am", "confused");
TextSpec t3 = new TextSpec(t1,t2,p1);
t3.setParagraph();
System.out.println(r.realiseDocument(t3));
```

results in the text

```
My cat likes fish, my dog likes bones and my horse likes grass. John is going to Tesco or Mary is going to Sainsburys. I am confused.
```

NLGKB interface

An NLGKB is an interface to a simple knowledge based which holds information needed for microplanning. An NLGKB consists of a taxonomy of classes, which have feature-value pairs (which should be inherited in the taxonomy from parent to child, unless overridden). An individual class is represented via the KBEntity interface (see below).

NLGKB and KBEntity are only used in microplanning, they are not needed if simplenlg is just used for realisation.

NLGKB can be implemented as Java classes (JavaKB) or as a Protégé KB (ProtegeKB); it can also be explicitly built in code (SimpleKB)

Key methods are

```
KBEntity getClass(String className);
boolean isAncestor(String className, String ancestorName);
```

KBEntity interface

A KBEntity is an entity in an NLG knowledge base. It is used both for classes and instances.

Key methods are

```
String getType();
Object getValue(String featureName);
```

JavaKB class

This implements an NLGKB as a set of Java classes. Features are just class fields.

All classes in a JavaKB must be in the same package, and they must extend (be subclasses of) simplenlg.JavaKBClass

Constructor is

JavaKB(String packageName); // param is name of package that contains the classes

Example:

```
package KBpackage;

public class pattern extends simplenlg.JavaKBClass {
        public int importance = 1;
        public String lex = "pattern"; }
public class spike extends pattern {
        public String lex = "spike";    // override lex feature;  importance is inherited
        public List potentialModifiers = Arrays.asList("momentary", "significant"); // can have list feature }

// in main program
NLGKB kb = new JavaKB("KBpackage");
KBEntity spikeClass = kb.getClass("spike");  // gets class named "spike"
                        // (actually returns instance of this class)
String spikeLex = spikeClass.getValue("lex"); // gets value of "lex" field
```

ProtegeKB class

This implements an NLGKB as a Protégé knowledge base. The knowledge base is constructed in Protégé and saved in the standard Protégé format. I've not tried this with Protégé-Owl.

If ProtegeKB is used, then protege.jar (from the protégé directory) must be included as a library in the Java build path. Note that it can take a noticeable amount of time to load a protégé KB (which is done when the constructor is called)

Constructor is

```
ProtegeKB(String projectFileName); // param is name of Protégé .pprj  file
```

Example:

```
NLGKB kb = new ProtegeKB(""ontology\\babytalk.pprj");
// once loaded, usage is identical to JavaKB example before
```

SimpleKB class

SimpleKBClass class

SimpleKB allows users to explicitly create an NLGKB.

Constructor is

SimpleKB();

Key SimpleKB methods (in addition to interface methods) are

SimpleKBClass newClass(String className);
SimpleKBClass newClass(String className, SimpleKBClass parent);
SimpleKBClass newClass(SimpleKBClass parent);

Key SimpleKBClass methods (in addition to interface methods) are

setFeatureValue(String featureName, Object value);

Example:

```
SimpleKB kb = new SimpleKB();
SimpleKBClass pattern = kb.newClass("pattern");
SimpleKBClass spike = kb.newClass("spike", pattern);
pattern.setFeatureValue("importance", 1);
pattern.setFeatureValue("lex", "pattern");
spike.setFeatureValue("lex", "spike");
spike.setFeatureValue ("potentialModifiers",Arrays.asList("momentary",
"significant"));

// once KB is set up, usage is as in JavaKB example
```

Lexicaliser class

The Lexicaliser converts instances in the knowledge base into linguistic structures that can be realised. More precisely, it generates a PhraseSpec for a KBEntity.

The simplenlg lexicaliser does this in a very simple way, by instantiating templates (either for a StringPhraseSpec or for components of an SPhraseSpec). The templates are defined in the KB, and they can refer to features of the entity being lexicalised. These instantiated templates can be post-processed (eg, to add tense/negated/passive/etc flags, and to add additional modifiers) by the application program if desired.

Templates consist of strings that can have feature references in them; these are in []. For example, if the template "a spike of importance [importance]" was associated with a spike class (see NLGKB examples), then [importance] would be replaced by the value of the importance feature of the entity being lexicalised. Eg, we would get "a spike of importance 1" if the entity has an importance feature whose value was 1.

More specifically, the following are allowed in [] in templates
[feature] - replaced by the value of feature in this entity
[feature1.feature2] – if the value of feature1 is class/entity X, this is replaced by the value of feature 2 in X
[$class] – replaced by the ontology class with this name
[$class.feature] – replaced by the value of feature for the named class
[#method] – replaced by the return value of the named Java method, called on the KBEntity
[X|format] – X must be one of the above constructs. format is a format string for Java's String.format, this is used to format the value being instantiated. format is either a literal format string starting with "%", or a reference to a feature (as above), whose value is a format string

If a template consists of a single reference to a feature whose value is a KBEntity, then a RefSpec for this entity is returned.

Constructor is

Lexicaliser(NLGKB kb); // parameter is the knowledge base used for lexicalisation

Key methods are

PhraseSpec lexicalise(KBEntity entity); // lexicalise an entity

Key KB features used in lexicalisation are

template – specifies template StringPhraseSpec

verb, subject, complement, modifier – specifies templates for components of SPhraseSpec

Example

```
SimpleKB kb = new SimpleKB();                          // create a KB
Lexicaliser lex = new Lexicaliser(kb);          // create lexicaliser based on KB
SimpleKBClass trendup = kb.newClass("trendup");  // class for upwards trend
trendup.setFeatureValue("verb", "increase");            // verb template
trendup.setFeatureValue("subject", "[channel.shortname]");  // subject
trendup.setFeatureValue("modifier", "to [endValue|%.0f]");   // modifier
SimpleKBClass heartrate = kb.newClass("heartrate");   // class for heart rate
heartrate.setFeatureValue("shortname", "HR");  // shortname of heartrate
SimpleKBClass trend1 = kb.newClass(trendup);        // instance of trend
trend1.setFeatureValue("channel", heartrate);        // trend channel = heartrate
trend1.setFeatureValue("endValue", 123.45);        // trend endValue = 123.45
PhraseSpec spec = lex.lexicalise(trend1);
```

spec is an SPhraseSpec which is realised as "HR increases to 123"

Design Rationale

Simplenlg is intended to be used in two ways

- realiser and (eventually) microplanner for Java-based NLG systems which construct relatively simple sentences (from a syntactic perspective); I am thinking particularly of data-to-text systems
- teaching tool for students learning about NLG

Our main goal is to automatic mundane and boring tasks which are relatively easy to automate, such as

- orthography: whitespace, punctuation absorption, pouring, lists
- morphology: inflected forms
- lexicon: access to info in decent-coverage lexicon
- simple grammar: sentence formation, agreement, verb groups
- template-based lexicalisation

These are all things that I personally would rather not worry about when building an NLG system.

Another key design goal is to allow the system to be used by people who have a basic understanding of grammar (eg, know what a subject is), but are not familiar with the details of any particular linguistic theory.

References

G Minnen, J Carroll, and D Pearce (2001). Applied Morphological Processing of English. *Natural Language Engineering* **7**: 207-223.

E Reiter and R Dale (2000). *Building Applied Natural Language Generation Systems.* Cambridge University Press.

http://specialist.nlm.nih.gov/ (Specialist lexicon)

Change Log

version 2

- This is quite different from v1. Essentially I’ve reduced the scope of what simplenlg is trying to do (eg, no more discourse relations), and at the same time tried to improve coverage, robustness, and usability of what remains in the reduced scope.

version 3

- Includes lexicaliser and NLGKB; hopefully the first step towards a proper microplanner
- (3.1) improved error handling (no functionality changes)
- (3.2) includes Albert Gatt's Lexicon package, with vastly improved morphological coverage. Also the system no longer attempts to infer whether words are plural, based on whether they end in "s" or not.
- (3.3) many bug fixes, Javadoc for API
- (3.4) many changes to the grammar, based on people's requests. Package also restructured into subpackages such as simplenlg.realiser
- (3.5) further changes to the grammer, including more flexible coordination. Better Javadoc for API, structured testing
- (3.6) support for questions, bug fixes
- (3.7) added DB lexicon, misc enhancements and bug fixes