Lexicon-lookup Component in *SProUT*

Jakub Piskorski
German Research Center for Artificial Intelligence
Stuhsatzenhausweg 3, 66123 Saarbrücken, Germany
piskorsk@dfki.de

VERSION JANUARY 2005
(SPROUT VERSION 4.0 AND HIGHER)

1 Introduction

SProUT deploys a component for performing a lexicon lookup[1]. It provides several standard operations for searching the lexicon, e.g., exact match search, retrieval of lexical information (part-of-speech, base form), etc. Further, several other operations like searching the longest/shortest prefix or suffix are provided, which might be of particular interest in the context of NLP. New lexica can be created via compiling valid *SProUT*-lexica in XML format. Furthermore, there are two ways of writing (reading) lexica to (from) files. The first method is based on the standard JAVA Serialization mechanism, and the second uses specific I/O format, which allows for more efficient processing. This rest of this paper briefly introduces the crucial issues concerning creating the lexica, using this lexicon-lookup component in *SProUT* and give some code examples for developers on how to integrate it in other applications.

2 Lexicon Construction

In order to construct a *SProUT* lexicon resource (in binary compressed format), three input files are needed. The first one contains lexicon entries. The second contains entity mapping for expansion of entities used in the lexicon entry file. And finally, the third file contains a corresponding and compatible type hierarchy for all morphological features (compiled via application of *flop* – a tool delivered with *SProUT* which allows for compiling type hierarchies).

[1] Standard trie data structure has been used for implementation of this component.

The lexicon entry file is an XML file, which simply contains the lexical entries. Each such entry is encoded as a disjunction of feature structures[2], where each of these feature structures (of type *morph*) represents a single reading of a given entry. The following piece of text illustrates the format of the lexicon entry file. In this example, there are three reading for the German word *evaluierten*.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE fullform_lexicon SYSTEM "german.dtd">
<fullform_lexicon>
……
<DISJ>
<FS type="morph">
<F name="SURFACE"><FS type="evaluierten"></FS></F>
<F name="STEM"><FS type="evaluieren"></FS></F>
<F name="POS"><FS type="adjective"></FS></F>
&inf11;
</FS>
<FS type="morph">
<F name="SURFACE"><FS type="evaluierten"></FS></F>
<F name="STEM"><FS type="evaluieren"></FS></F>
<F name="POS"><FS type="verb"></FS></F>
&inf12;
</FS>
<FS type="morph">
<F name="SURFACE"><FS type="evaluierten"></FS></F>
<F name="STEM"><FS type="evaluieren"></FS></F>
<F name="POS"><FS type="verb"></FS></F>
&inf13;
</FS>
</DISJ>
……
</fullform_lexicon>
</xml>
```

The entity-mapping file includes the definition of entities, and will be used for expanding the entities in the lexicon entry file. A corresponding fragment of the entity mapping for our example of lexicon file would look like follows.

```
<?xml version='1.0' encoding='ISO-8859-1'?>
……
<!ENTITY inf11  '<F name="INFL"><FS type="infl_adjective"><F name="CASE_ADJECTIVE"><FS
type="acc_dat_gen_nom"></FS></F><F name="STTS_OPEN_ADJECTIVE"><FS type="adja"></FS></F><F
name="GENDER_ADJECTIVE"><FS type="fem_masc_neutrum"></FS></F><F name="NUMBER_ADJECTIVE"><FS
type="plural_singular"></FS></F><F name="SPELLING_ADJECTIVE"><FS type="unchanged"></FS></F><F
name="DEGREE_ADJECTIVE"><FS type="pos"></FS></F></FS></F>' >
<!ENTITY inf12  '<F name="INFL"><FS type="infl_verb"><F name="PERSON_VERB"><FS
type="1_3"></FS></F><F name="PARTICLE_VERB_VERB"><FS type="none"></FS></F><F
name="TENSE_VERB"><FS type="imperfect"></FS></F><F name="STTS_OPEN_VERB"><FS
type="vvfin"></FS></F><F name="NUMBER_VERB"><FS type="plural"></FS></F><F
name="SPELLING_VERB"><FS type="unchanged"></FS></F><F name="MODE_VERB"><FS
type="indicative"></FS></F><F name="VFORM_VERB"><FS type="fin"></FS></F></FS></F>' >
<!ENTITY inf13  '<F name="INFL"><FS type="infl_verb"><F name="PERSON_VERB"><FS
type="1_3"></FS></F><F name="PARTICLE_VERB_VERB"><FS type="none"></FS></F><F
name="TENSE_VERB"><FS type="imperfect"></FS></F><F name="STTS_OPEN_VERB"><FS
type="vvfin"></FS></F><F name="NUMBER_VERB"><FS type="plural"></FS></F><F
name="SPELLING_VERB"><FS type="unchanged"></FS></F><F name="MODE_VERB"><FS
type="subjunctiveii"></FS></F><F name="VFORM_VERB"><FS type="fin"></FS></F></FS></F>' >
……
```

[2] We assume that the reader is familiar with encoding of feature structures in XML format

How the lexicon entry and entity mapping file are created is up to the user. Currently, we have automatically created such resources from MMorph database. However, there are no limits where the data comes from.

Please note that, there are following prerequisites which must be fulfilled in order one can successfully compile the lexicon resources into binary compressed format and utilize them in *SProUT* or elsewhere:

(1) Each feature structure of type *morph* encoding single reading must include the attributes `POS` (part-of-speech), `STEM`, and `SURFACE`. Additionally, if inflection information is encoded, then `INFL` attribute should be used for this purpose.

(2) Each type and attribute which appears in the lexicon entry and entity mapping file must be defined in the type hierarchy, and the attributes must be appropriate for the corresponding types in the input data. Finally, each attribute's value must be appropriate for this attribute.

(3) Double entries should be avoided, i.e., all readings for a given word form should be grouped together into one disjunction of feature structures. If this is not the case, then the last entry, overwrites the previous ones.

If the conditions (1) and (2) are not fulfilled, then the compilation process will be aborted. If the condition (3) is not fulfilled, then warnings will be generated.

The compilation can be done either via application of the methods **`de.dfki.lt.sprout.runtime.morphology.lexicon.Lexicon.construct()`**, or **`de.dfki.lt.sprout.runtime.morphology.lexicon.Lexicon.constructBlockwise()`**. Please see, the Java documentation for details. Alternatively, the following script might be used for the same purpose[3]:

```
java -Xmx640M -cp <LibPath> de.dfki.lt.sprout.runtime.morphology.lexicon.CompileLexicon
     <LexiconEntryFile> <EntitiesFile> <TypeHierarchyFile> <OutputFile> <Lang>

      <LibPath> specifies: (a) the path to the directory containing the SProUT Java archive
                or a root directory containing the compiled CompileLexicon class,
                and (b) the path to the log4j-1.2.8.jar archive

      <LexiconEntryFile> a path to the lexicon entry file

      <EntitiesFile> a path to the file containing the entity mapping

      <TypeHierarchyFile> a path to the file containing the type hierarchy

      <OutputFile> specifies the output file (we use 'slx' extension for denoting compressed
                   lexicon files)
```

[3] For internal DFKI staff: The latest versions of all `jar` archives and external libraries which are used by *SProUT* can be found in DFKIs internal directory /project/cl/sprout/test/resource/lib. External users will find these libraries in their main *SProUT* directory.

<LanguageID> specifies the language (for German use the tag 'german', and for Durch please use the tag 'nl', for other languages feel free to use any naming convention)

3 Running the Sentence Boundary Recognizer

When using the lexicon-lookup component within the *SProUT* integrated development environment, you have to choose the option *Menu -> Tools -> Configure components* which leads you to the configuration window in which you can launch an instance of such component in the similar way to installing other processing resources. The only field, which has to be specified is the path to the file containing the compressed binary representation of the lexicon.

In order to use the lexicon-lookup component by the developers independently, the following piece of code (in a simplified form) could be used, which illustrates all indispensable steps that have to be undertaken in order to initialize a lexicon-lookup component and to perform a simple look-up of a given word.

```
import java.io.*;
import de.dfki.lt.tfs.*;
import de.dfki.lt.sprout.runtime.morphology.lexicon.*;

// For storing the result of the morphological analysis
Disjunction Disj;

// Initialization

// Define and initialize a new grammar object (type hierarchy)
ShUG grammar = new ShUG("typeHierarcht.grm");

// Create a lexicon object
Lexicon L = new Lexicon();

// Set the basic logger
L.setBasicLogger();

// Read the lexicon from file
L.readFromFile("LexiconFile.slx",grammar);

// Testing

// Define input word
String word = "house";

// Search in lexicon
Disj = (Disjunction)L.search(word);

// Display the results
System.out.print("Testing word: " + word);
if(Disj!=null)
 { System.out.println(); Disj.printFS(); }
else
 { System.out.println("No Entry found"); }
```

For further details, please see the Java documentation in the APPENDIX A.

APPENDIX A - JAVA DOCUMENTATION

JAVA DOCUMENTATION FOR THE CLASS GeneralLexicon

de.dfki.lt.sprout.runtime.morphology.lexicon
Interface GeneralLexicon

All Known Implementing Classes:
Lexicon

public interface **GeneralLexicon**

This is the general class for Lexicon-like data structure.

Method Summary

FS	**search**(java.lang.String w) Returns lexical information (a single feature structure or disjunction of feature structures) associated with the input word

Method Detail

search

public FS **search**(java.lang.String w)

Returns lexical information (a single feature structure or disjunction of feature structures) associated with the input word

Parameters:
w - word input string

JAVA DOCUMENTATION FOR THE CLASS Lexicon

de.dfki.lt.sprout.runtime.morphology.lexicon
Class Lexicon

java.lang.Object
└ **de.dfki.lt.sprout.runtime.morphology.lexicon.Lexicon**

All Implemented Interfaces:
GeneralLexicon, java.io.Serializable

public class **Lexicon**
extends java.lang.Object
implements GeneralLexicon, java.io.Serializable

This class implements Lexicon-like data structure. It provides several operations for searching different type of lexical information. Beyond standard operations several prefix/sufix operations are provided. Further, new lexica can be created by converting valid SproUT-lexica in the XML Format. There are two ways of writing(reading) lexica to(from) files. The first method is based on the standard JAVA Serialization mechanism, and the second uses specific I/O format which allows for more efficient processing.

See Also:
Serialized Form

Field Summary

`static java.lang.String`	**INFL_DEF** A constant which specifies the name of the `inflection` attribute.
`static java.lang.String`	**MORPH_DEF** A constant which specifies the name of the type for representing morphological information.
`static java.lang.String`	**POS_DEF** A constant which specifies the name of the `part-of-speech` attribute.
`static java.lang.String`	**STEM_DEF** A constant which specifies the name of the `stem` attribute.
`static java.lang.String`	**SURFACE_DEF** A constant which specifies the name of the `surface` attribute.

Constructor Summary

Lexicon`()` Creates an empty Lexicon (default constructor).

Method Summary

`java.util.ArrayList`	**CheckConsistency**`(ShUG grammar)` This function performs a consistency check in order to test if all feature structure types and attributes used by the lexicon (also types appearing in the lexicon entries) component are present in the given type hierarchy.

java.util.ArrayList	**checkConsistencyDeeper**(ShUG grammar) This function performs a special consistency check in order to test if all types/attribues appearing in the lexicon entries are present in the given type hierarchy and if they are appropriate.
boolean	**configureLogger**(java.lang.String LoggerPropertyFile) Configure the logger according to the Logger property file
void	**construct**(java.lang.String filename, java.lang.String Entitiesfile, java.lang.String GrammarFile, java.lang.String OutputFile, java.lang.String Lang, boolean useSerialization) Constructs a compressed binary representation of an input lexicon in the SProUT-specific XML Format.
void	**constructBlockwise**(java.lang.String filename, java.lang.String Entitiesfile, java.lang.String GrammarFile, java.lang.String OutputFile, java.lang.String Lang, boolean useSerialization) Constructs a compressed binary representation of an input lexicon in the SProUT-specific XML Format.
boolean	**contains**(java.lang.String word) Returns a boolean value indicating if the input word is included in the current lexicon.
java.lang.String[]	**get_pos_information**(java.lang.String word) Returns part-of-speech information for the input word.
java.lang.String[]	**get_stems**(java.lang.String word) Returns all stems associated with the input word.
ShUG	**getGrammar**() Returns the grammar associated with the lexicon
java.lang.String	**getLanguage**() Returns the language associated with the lexicon.
int	**getNumberEntries**() Returns number of entires currently stored in the lexicon.
java.lang.String	**longestPrefix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the longest prefix of a given string included in the lexicon.
java.lang.String	**longestSufix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the longest sufix of a given string included in the lexicon.
boolean	**read**(java.lang.String LexiconFile, ShUG Grammar) Reads a lexicon from a given file.
boolean	**readFromFile**(java.lang.String filename, ShUG grammar) Reads a lexicon from a given file (in Lexicon specific format).
void	**readFromStream**(java.nio.ByteBuffer b) Reads the internal representation of the lexicon from an input byte buffer.

static Lexicon	**readLexicon**(java.lang.String LexiconFile, ShUG Grammar) Reads a lexicon from the file.
boolean	**saveToFile**(java.lang.String filename) Writes the lexicon to a given file (in Lexicon specific format).
FS	**search**(java.lang.String word) Returns lexical information associated with the input word.
boolean	**search**(java.lang.String word, java.util.ArrayList alList) Specific searech function used by SProUT (do not use it)
FS	**searchLongestPrefix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the longest prefix of a given string included in the lexicon.
FS	**searchLongestSufix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the longest sufix of a given string included in the lexicon.
FS	**searchShortestPrefix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the shortest prefix of a given string included in the lexicon.
FS	**searchShortestSufix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the shortest sufix of a given string included in the lexicon.
void	**setBasicLogger**() Activates the default basic logger (standard output)
java.lang.String	**shortestPrefix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the shortest prefix of a given string included in the lexicon.
java.lang.String	**shortestSufix**(java.lang.String word) Performs a lookup in the lexicon in order to search for the shortest sufix of a given string included in the lexicon.
void	**writeToStream**(java.io.DataOutputStream s) Writes the internal representation of the lexicon into an output stream.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

POS_DEF

public static final transient java.lang.String **POS_DEF**

A constant which specifies the name of the `part-of-speech` attribute.

See Also:
Constant Field Values

STEM_DEF

public static final transient java.lang.String **STEM_DEF**

A constant which specifies the name of the `stem` attribute.

See Also:
Constant Field Values

SURFACE_DEF

public static final transient java.lang.String **SURFACE_DEF**

A constant which specifies the name of the `surface` attribute.

See Also:
Constant Field Values

INFL_DEF

public static final transient java.lang.String **INFL_DEF**

A constant which specifies the name of the `inflection` attribute.

See Also:
Constant Field Values

MORPH_DEF

public static final transient java.lang.String **MORPH_DEF**

A constant which specifies the name of the type for representing morphological information.

See Also:
Constant Field Values

Constructor Detail

Lexicon

public **Lexicon**()

Creates an empty Lexicon (default constructor).

Method Detail

getGrammar

public ShUG **getGrammar**()

Returns the grammar associated with the lexicon

Returns:

the grammar (ShUG object) associated with the lexicon

---

getNumberEntries

public int **getNumberEntries**()

Returns number of entires currently stored in the lexicon.

Returns:

the number of entries stored in the lexicon

---

contains

public boolean **contains**(java.lang.String word)

Returns a boolean value indicating if the input word is included in the current lexicon.

Parameters:

`word` - input string

Returns:

`true` if the input word is present in the lexicon, or `false` otherwise.

---

search

public FS **search**(java.lang.String word)

Returns lexical information associated with the input word.

Specified by:

search in interface GeneralLexicon

Parameters:

`word` - input string

Returns:

a feature structure or disjunction of feature structures which represent the lexical information associated with the input word, or `null` if the input word is not contained in the lexicon

---

configureLogger

public boolean **configureLogger**(java.lang.String LoggerPropertyFile)

Configure the logger according to the Logger property file

Parameters:

`LoggerPropertyFile` - the property file used for configuring the logger

Returns:

`true` if the configuration was successful, or `false` otherwise.

---

setBasicLogger

public void **setBasicLogger**()

Activates the default basic logger (standard output)

---

searchLongestPrefix

public FS **searchLongestPrefix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the longest prefix of a given string included in the lexicon.

Parameters:

`word` - input string

Returns:

the lexical information associated with the the longest prefix of the input string found in the lexicon, or `null` if no such prefix was found.

---

longestPrefix

public java.lang.String **longestPrefix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the longest prefix of a given string included in the lexicon.

Parameters:

`word` - input string

Returns:

the longest prefix of the input string found in the lexicon, or `null` if no such prefix was found.

---

searchShortestPrefix

public FS **searchShortestPrefix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the shortest prefix of a given string included in the lexicon.

Parameters:

`word` - input string

Returns:

the lexical information associated with the the shortest prefix of the input string found in the lexicon, or `null` if no such prefix was found.

---

shortestPrefix

public java.lang.String **shortestPrefix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the shortest prefix of a given string included in the lexicon.

Parameters:
`word` - input string
Returns:
the shortest prefix of the input string found in the lexicon, or `null` if no such prefix was found.

---

searchShortestSufix

public FS **searchShortestSufix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the shortest sufix of a given string included in the lexicon.
Parameters:
`word` - input string
Returns:
the lexical information associated with the the shortest sufix of the input string found in the lexicon, or `null` if no such sufix was found.

---

shortestSufix

public java.lang.String **shortestSufix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the shortest sufix of a given string included in the lexicon.
Parameters:
`word` - input string
Returns:
the shortest sufix of the input string found in the lexicon, or `null` if no such sufix was found.

---

searchLongestSufix

public FS **searchLongestSufix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the longest sufix of a given string included in the lexicon.
Parameters:
`word` - input string
Returns:
the lexical information associated with the the longest sufix of the input string found in the lexicon, or `null` if no such sufix was found.

---

longestSufix

public java.lang.String **longestSufix**(java.lang.String word)

Performs a lookup in the lexicon in order to search for the longest sufix of a given string included in the lexicon.

Parameters:

`word` - input string

Returns:

the longest sufix of the input string found in the lexicon, or `null` if no such sufix was found.

---

get_stems

public java.lang.String[] **get_stems**(java.lang.String word)

Returns all stems associated with the input word.

Parameters:

`word` - input string

Returns:

an array of stems, or`null` if the word is not contained in the lexicon.

---

get_pos_information

public java.lang.String[] **get_pos_information**(java.lang.String word)

Returns part-of-speech information for the input word.

Parameters:

`word` - input string

Returns:

an array of part-of-speech tags, or`null` if the word is not contained in the lexicon.

---

writeToStream

public void **writeToStream**(java.io.DataOutputStream s)
throws java.io.IOException

Writes the internal representation of the lexicon into an output stream.

Parameters:

`s` - an output stream

Throws:

`java.io.IOException`

---

readFromFile

public boolean **readFromFile**(java.lang.String filename,
ShUG grammar)

Reads a lexicon from a given file (in Lexicon specific format).

Parameters:

`filename` - is the name of the input file

`grammar` - is the name of the associated grammar

Returns:
`true` if this operation was succesfull, and `false` if the operation failed.

---

saveToFile

public boolean **saveToFile**(java.lang.String filename)

Writes the lexicon to a given file (in Lexicon specific format).

Parameters:
`filename` - is the name of the output file

Returns:
`true` if this operation was succesfull, and `false` if the operation failed.

---

readFromStream

public void **readFromStream**(java.nio.ByteBuffer b)

Reads the internal representation of the lexicon from an input byte buffer.

Parameters:
`b` - an input byte buffer

---

constructBlockwise

public void **constructBlockwise**(java.lang.String filename,
java.lang.String Entitiesfile,
java.lang.String GrammarFile,
java.lang.String OutputFile,
java.lang.String Lang,
boolean useSerialization)

Constructs a compressed binary representation of an input lexicon in the SProUT-specific XML Format. This can take several minutes. Note, that this method reads the lexicon input files piece by piece in contrary to the method `construct`.

Parameters:
`filename` - input lexicon entry file (in XML format)
`Entitiesfile` - a file containing entity mapping which are used in the lexicon file (in XML format)
`GrammarFile` - is the name of the file containing the associated grammar (type hierarchy)
`OutputFile` - is the file to which the binary representation will be written to
`Lang` - is the language information associated with the lexicon (If intend to use the compiled resources in SProUTm the use for the tag 'german', for Dutch 'nl', and for any other languages any tag you like
`useSerialization` - is a flag which should be set to `true` if standard JAVA serialization mechanism is to be used for writing and reading this lexicon (for SProUT: set this flag to `false`).

IMPORTANT: There are several prerequisites which must/should be fulfilled

(1) All types/attributes which appear in the lexicon source files must be present in the type hierarchy (GrammarFile) and they must be appropriate. If this condition does not apply, then the compilation process will be aborted

(2) Double entries should be avoided, otherwise the last one overwrites the previous ones

---

construct

public void **construct**(java.lang.String filename,
java.lang.String Entitiesfile,
java.lang.String GrammarFile,
java.lang.String OutputFile,
java.lang.String Lang,
boolean useSerialization)

Constructs a compressed binary representation of an input lexicon in the SProUT-specific XML Format. This can take several minutes. Note, that this method reads an entire XML File into memory, in contrary to the method `construct`.

Parameters:

`filename` - input lexicon entry file (in XML format)

`Entitiesfile` - a file containing entity mapping which are used in the lexicon file (in XML format)

`GrammarFile` - is the name of the file containing the associated grammar (type hierarchy)

`OutputFile` - is the file to which the binary representation will be written to

`Lang` - is the language information associated with the lexicon (If intend to use the compiled resources in SProUTm the use for the tag 'german', for Dutch 'nl', and for any other languages any tag you like

`useSerialization` - is a flag which should be set to `true` if standard JAVA serialization mechanism is to be used for writing and reading this lexicon (for SProUT: set this flag to `false`).

IMPORTANT: There are several prerequisites which must/should be fulfilled

(1) All types/attributes which appear in the lexicon source files must be present in the type hierarchy (GrammarFile) and they must be appropriate. If this condition does not apply, then the compilation process will be aborted

(2) Double entries should be avoided, otherwise the last one overwrites the previous ones

---

read

public boolean **read**(java.lang.String LexiconFile,
ShUG Grammar)

Reads a lexicon from a given file. Note, that this method is based on the standard JAVA serialization mechanism.

Parameters:

`Grammar` - is the associated grammar

Returns:

`true` if this operation was succesfull, and `false` if the operation failed.

---

readLexicon

public static Lexicon **readLexicon**(java.lang.String LexiconFile,
ShUG Grammar)

Reads a lexicon from the file. Note, that this method is based on the standard JAVA serialization mechanism.

Parameters:

`Grammar` - is the associated grammar

Returns:

a Lexicon object if this operation was succesfull, and `null` if the operation failed.

---

getLanguage

public java.lang.String **getLanguage**()

Returns the language associated with the lexicon.

Returns:

the language associated with the lexicon

CheckConsistency

public java.util.ArrayList **CheckConsistency**(ShUG grammar)

This function performs a consistency check in order to test if all feature structure types and attributes used by the lexicon (also types appearing in the lexicon entries) component are present in the given type hierarchy.

Parameters:

`grammar` - object containing a type hierarchy for testing the consistency

Returns:

an array list of String containing all inconsistencies. If no incosistencies were found, the array list will be empty.

---

checkConsistencyDeeper

public java.util.ArrayList **checkConsistencyDeeper**(ShUG grammar)

This function performs a special consistency check in order to test if all types/attribues appearing in the lexicon entries are present in the given type hierarchy and if they are appropriate.

Parameters:

`grammar` - object containing a type hierarchy for testing the consistency

Returns:

an array list of errors. If no incosistencies were found, the array list will be empty.

---

search

public boolean **search**(java.lang.String word,
java.util.ArrayList allList)

Specific searech function used by SProUT (do not use it)
