INFOGISTICS' NLProcessor
TEXT PROCESSING TOOLKIT
[DRAFT VERSION]

infogistics

December 2000

Tel: +44 (131) 650 4632
Email info@infogistics.com
Web: http://www.infogistics.com/

TABLE OF CONTENTS

I NLProcessor – general description

II NLProcessor output
III.1 NLProcessor output – tokenization
III.2 NLProcessor output – POS tagging
III.3 NLProcessor output – syntactic chunking

III NLProcessor – standalone tool

IV. NLProcessor – server mode

V. NLProcessor resources – fine tuning the performance

VI. NLProcessor SDK
VI.1 Calling NLProcessor from your code
VI.2 Complete API Specification

VII. Java API : package com.infogistics.nlprocessor

VII.1 class TSConnection (extends Object)

VII.2 class NLProcessor (extends Object)

VII.3 Working with XML tokenization
VII.3.1 class Token (implements Object)
VII.3.2 class Chunk (implements Object)
VII.3.3 class TokenIterator (implements Iterator)

VII.3.4 Example

I NLProcessor – general description

NLProcessor by Infogistics is a successor for a set of Natural Language Processing technologies developed in the 1990s at the University of Edinburgh. NLProcessor is an engine which handles so-called "low-level" text processing routines: tokenization, capitalized word normalization, sentence segmentation, part-of-speech tagging and syntactic chunking which are necessary steps in building many kinds of text handling applications.

Electronic text is essentially just a sequence of characters some of which are content characters, such as letters of an alphabet, numbers, punctuation, etc., and others are control and formatting (typesetting) characters such as whitespace, newlines, etc. Naturally, before any real text processing is to be done, text (the sequence of characters) needs to be segmented at least into linguistic units such as words, punctuation, numbers, alphanumeric, etc. This process is called tokenization, and segmented units are called word-tokens. Tokenization is a relatively straightforward task for languages like English and other languages, where words are delimited by blank spaces and punctuation. However, there are cases where tokens are written with no explicit boundaries between them and some cases where what seems to be two tokens (i.e. delimited by a whitespace) in fact form one and vice versa.

In mixed-case texts capitalized words usually denote proper names -- names of organizations, locations, people, artifacts, etc., but there are some positions in the text where capitalization is expected. Such mandatory (ambiguous) positions include the first word in a sentence, words in all-capitalized titles or table entries, a capitalized word after a colon or open quote, the first capitalized word in a list-entry, etc. Capitalized words in these and some other positions present a case of ambiguity -- they can stand for proper names as in *``White later said...''* or they can be just capitalized common words as in *``White elephants are...''* Obviously, this distinction is important for almost all kinds of text analysis.

Segmenting text into sentences in most cases is a simple matter -- a period, an exclamation mark or a question mark usually signal a sentence boundary. However, there are cases when a period denotes a decimal point or is a part of abbreviation and thus it does not signal a sentence break. Furthermore, an abbreviation itself can be the last token in a sentence in which case its period acts at the same time as part of this abbreviation and as the end-of-sentence indicator (fullstop). Therefore, segmentation of sentences can present some unexpected difficulties, which need to be addressed.

NLProcessor approaches these tasks through a proprietary on-line learning algorithm applied in conjunction with pre-built resources. First, it scans the text and learns important regularities of the way it is composed. Then it applies these regularities, together with information from its resources, to identify word and sentence boundaries, as well as to predict whether a capitalized word stands for a proper name or whether it can be safely downcased. NLProcessor outputs this information by directly marking text with XML markup: tokens are represented as "W" elements, word-class information is provided in their "T" attribute and sentences are marked with "S" elements. For example,

```
<S><W  T=w>First</W><W T=P>,</W> <W T=W>John</W> <W T=w>needs</W>
    <W T=N>25</W> <W T=w>kg.</W> <W T=w>of</W> <W T=w>sand</W>
    <W T=".">.</W></S>
```

Here, each word token is marked as a W element. T=w stands for a standard word and T=W means a proper name, so even though the word "First" was written capitalized it was classified as a standard word while the word "John" was classified as a proper name. We also see that token "kg." includes a period and therefore is an abbreviation whereas the period after "sand" is a separate token and signals the end of the sentence. Whitespace separation between tokens is left untouched.

Apart from standard words, proper names, punctuation, and numerals, NLProcessor is also equipped to process several specialized types of tokens: email addresses, URL and file names, dates, telephone and fax numbers and some measure expressions.

The task of POS-tagging is to assign part-of-speech tags to words reflecting their syntactic category. Often, a word can belong to different syntactic categories in different contexts. For instance, the string "books" can have two readings: in the sentence *he books tickets* the word "books" is a third person singular verb, but in the sentence *he reads books* it is a plural noun. A POS-tagger tries to determine possible readings for a word, and to assign the right reading given the context. This information is then represented in a attribute of a token.

NLProcessor also includes a syntactic chunker or partial parser. It uses the part-of-speech information provided by the tagger and employs a mildly context-sensitive grammars to detect boundaries of syntactic groups. The chunker leaves all previously added information in the text and creates a structural element which includes the words of the chunk. Currently it is capable of recognizing boundaries of simple noun and verb groups :

<NG>The most important man**</NG>** in **<NG>**our little group**</NG>**.
<NG>The red book**</NG>** which I **<VG>**bought**</VG>** **<NG>**yesterday**</NG>**.

NLProcessor can be used as a standalone product or through its SDK it can be integrated into third-party applications. It currently works with several different dialects of English (American, British, Canadian, Australian, Hong-Kong). Apart from its core engine, NLProcessor also comes with a toolkit for building resources for the segmentation of customer-defined tokens.

NLProcessor is available for all major platforms (WIN32, Linux, Solaris, SunOS..) and can be used as a standalone product or it can be integrated into your application as a dynamic library or COM/CORBA object. Compatible compilers: Microsoft Visual C++ 5.0 (and higher) on WIN32, gcc 2.5 and higher on Linux, Solaris and SunOS. NLProcessor also comes with Java API and can be easily integrated into Java applications.

II NLProcessor output

As was briefly discussed above, NLProcessor segments words, sentences and syntactic groups and marks this information, together with word-class information, using XML. It does this in a “non-destructive” mode, so if NLProcessor-generated annotation is removed, the original text will remain.

III.1 NLProcessor output – tokenization

By default, NLProcessor marks-up tokens in XML as “W” elements and puts information about the token-class into its “T” attribute. This, however, is parametrizable and can be changed to use different element labels and attributes. This is also true for sentence markup – the default sentence element label is S but it can be changed to any other valid element name.

There are several classes of tokens recognized by NLProcessor:

Flag	Meaning	Explanation
w	Regular word	Such words are written lowercased in the middle of a sentence and capitalized in sentence-starting positions.
W	Proper name	Such words are written capitalized regardless whether they are sentence starting or used in the middle of a sentence.
N	Numerical	This includes simple (e.g. 25) and complex (e.g. 25.890,78) formats.

O	Ordinal	Tokens like 3rd, 12 th, etc.
P	Punctuation	Commas, semicolons, periods
.	Sentence end	A period, question mark or exclamation mark
Email	Email	Email addresses e.g. a.mikheev@infogistics.com
Date	Date	Numerical date expressions e.g. 15/May/64, 15.05.1964, May-15-1964, etc.
URL	URL and files	URL and filenames e.g. www.infogistics.com
TEL	Telephone	Telephone and FAX numbers: e.g. +44 0131 247 5650 or 1-800-232 SAVE

Abbreviations are marked up including their trailing periods e.g. "<W T=W>Dr.</W>". When an abbreviation is the last token of a sentence its period serves also as a sentence delimiter. In this case it is tokenized together with the abbreviation but also a virtual fullstop token is added to the sentence. This virtual fullstop does not have character content but it has its attribute set to indicate that it is sentence delimiter:

<W T=w>etc.</W>**<W T=".”></W>**</S>

NLProcessor also marks up capitalized words in ambiguous positions, i.e. in positions where it thinks that capitalization is expected (sentence start, all-capitalized titles and sentences, etc.) by setting the attribute "S" to "Y":

<W T=w S=Y>**The**</W> <W T=w>**weather**</W> <W T=w>**in**</W> <W T=W>**Leeds**</W><W T=".">.</W>

NLProcessor can also output its confidence for a capitalized word to be classifies as "w" or "W". By default it uses the L attribute: e.g. <W T=W S=Y L=95>White</W> means that the word "White" has been classified as a proper name with confidence 95%.

NLProcessor can also segement sentences and proper titles

<S><W>I</W> <W>like</W> <TITLE><W>The</W> <W>Phantom</W> <W>of</W> <W>the</W> <W>Opera</W></TITLE> <W>.<W></S>

III.2 NLProcessor output – POS tagging

NLProcessor also can assign part-of-speech tags to words:

<W C=DT>**The**</W> <W C=NN>**weather**</W> <W C=IN>**in**</W> <W C=NNP>**Leeds**</W><W C=".">.</W>

here the word "The" was tagged as a determiner (DT), the word "weather" as a noun (NN), the word "in" as a preposition (IN) and the word "Leeds" as a proper noun (NNP). NLProcessor's tagger is trained on publicly available corpora using extended Penn Treebank tagset. Here is full list of tags which can be assigned to words:

Extended Penn Treebank Tag-Set (open class categories)

POS Tag	Description	Example
JJ	adjective	green
JJP	adjective, proper name	American
JJR	adjective, comparative	greener
JJS	adjective, superlative	greenest
RB	adverb	however, usually, naturally, here, good
RBR	adverb, comparative	better
RBS	adverb, superlative	best
NN	common noun	table
NNS	noun plural	tables
NNP	proper noun	John
NNPS	plural proper noun	Vikings
VB	verb base form	take
VBD	verb past	took
VBG	gerund	taking
VBN	past participle	taken
VBP	verb, present, non-3d	take
VBZ	verb present, 3d person	takes
FW	foreign word	d'hoevre

Extended Penn Treebank Tag-Set (closed class categories)

POS Tag	Description	Example
CD	cardinal number	1, third
CC	coordinating conjunction	and
DT	determiner	the
EX	existential there	*there* is
IN	preposition/subordinating conjunction	in, of, like
LS	list marker	1)
MD	modal	could, will
PDT	predeterminer	*both* the boys
POS	possessive ending	friend*'s*
PRP	personal pronoun	I, he, it
PRP$	possessive pronoun	my, his
RP	particle	give *up*
TO	to (both "to go" and "to him")	*to* go, *to* him
UH	interjection	uhhuhhuhh
WDT	wh-determiner	which
WP	wh-pronoun	who, what
WP$	possessive wh-pronoun	whose
WRB	wh-abverb	where, when

III.3 NLProcessor output – syntactic chunking

The results of syntactic chunking is grouping of words into noun groups and verb groups. By default noun groups marked up in text as NG elements and verb groups as VG elements:

```
<S><NG><W C=DT>The</W> <W C=NN>weather</W></NG> <W C=IN>in</W> <NG><W C=NNP>Leeds</W></NG> <VG><W
C=VBZ>is</W> <W C=RB>icurrently</W>  <W C=VBG>changing</W></VG> <W C=”.”>.</W></S>
```

here we see two noun groups “The weather” and “Leeds” and a verb group “is currently changing”.

III NLProcessor – standalone tool

The standalone version of NLProcessor works both with plain ASCII texts and with texts marked up in XML.

To call NLProcessor in standalone mode, you need to specify a resource file, which contains pointers to necessary resources (grammars, lexicons, statistical models, etc.). A standard resource file is provided with the NLProcessor installation, but you can also develop your own resources for your own domain and then create your own customized resource file.

Here is a typical call to The NLProcessor which uses the standard resources to process text from the file “file_to_process”, which is an ASCII file. We assume that your NLProcessor installation is in C:\NLP on a Win32 platform:

```
    >>nlproc  c:\NLP\Resource\resource.spc  file_to_process
```

this call will apply default settings and will tokenize text with “W” elements. To markup sentences in the text use –sent ELEMENT_LABEL option:

```
   >>nlproc -sent S c:\NLP\Resource\resource.spc  file_to_process
```

this call will tokenize text with “W” elements and segment sentences as “S” elements. To segment titles in the text use –title TITLE_LABEL option. In this case titles such as “Phantom of the Opera” will be identified and tokenized in the text.

To generate part-of-speech information for words you can use –tag ATTR_NAME options which specifies name of the attribute of a where POS tag should be placed. Here is an example of assigning C attribute with POS information:

```
>>nlproc  -tag C c:\NLP\Resource\resource.spc  file_to_process
```

Similarly you can request verb and noun group processing by setting –vg ELEMENT_LABEL and –ng ELEMENT_LABEL. Here is an example of tagging noun groups as NG and tagging verb groups as VG:

```
>>nlproc  -vg VG –ng NG c:\NLP\Resource\resource.spc  file_to_process
```

When you run NLProcessor over files with XML markup you need to specify which elements of the XML structure represent documents and which sections of the document to process. This is done by passing two access queries: –qd (for documents) –qs (for sections) options:

```
  >>nlproc  -qd “.*/DOC” –qs  “.*/P” c:\NLP\Resource\resource.spc  file_to_process.xml

    –qd  “.*/DOC”  -- all DOC elements should be treated as documents
    -qs     “.*/P”      -- text in P elements which are under DOC elements should be processed
```

If you don't have paragraphs marked up in the documents you should use the command line options: –qs "." meaning "get text from the parent element" which in this case is the DOC element. The ".*" prefix in the access queries means "anywhere under". So ".*/DOC" accesses all DOC elements in the file and ".*/P" accesses all P elements under DOC elements. One can specify more detailed access queries:
".*/DOC[id=1]" – process all DOCs with id attribute set to 1.
"/CORPUS/DOC" – process DOC elements which are directly under root CORPUS element.
For further information and query syntax see the LTXML Query Language document.

To change default markup settings use –mark ELEMENT_LABEL to mark tokens (default "W") and -class ATTRIBUTE_NAME (default "T") to markup word-class information.
Here is the summary of all command-line options:

Option	Argument	Explanation
-sent	<ELEMENT_LABEL>	Mark-up sentences with element ELEMENT_LABEL
-title	<ELEMENT_LABEL>	Markup titles with element ELEMENT_LABEL
-qd	<XML_Query>	Process XML input where documents can be accessed by this query
-qs	<XML_Query>	Process XML input where text in documents can be accessed by this query
-mark	<ELEMENT_LABEL>	Mark-up tokens with ELEMENT_LABEL (default W)
-class	<ATTRIBUTE_NAME>	Put token class info into attribute with this name (default T)
-tag	<ATTRIBUTE_NAME>	Put POS class info into attribute with this name.
-sent	<ELEMENT_LABEL>	Segment sentence into ELEMENT_LABEL els.
-title	<ELEMENT_LABEL>	Segment title into ELEMENT_LABEL elements
-ng	<ELEMENT_LABEL>	Segment noun groups into ELEMENT_LABEL elements
-vg	<ELEMENT_LABEL>	Segment verb groups into ELEMENT_LABEL elements

IV. NLProcessor – server mode

In addition to the stand-alone tools and SDK API, NLProcessor can also be delivered as a service within Infogistics' TextServer. The TextServer is architected as a TCP/IP-based service which communicates with an application via an efficient CORBA-like protocol. It comes with a COM interface which integrates Infogistics' technology into this popular Microsoft platform. It also comes with a Java API which makes it straightforward to use text processing functionality from Java applications.

Starting the TextServer involves simply starting a daemon process. Doing this is very straightforward:

```
 >> IGTextServer  -port PORT_NUMBER  <resource-file>
```

For example,

```
>> IGTextServer  -port  2222  c:\NLP\Resources\resource.spc
```

where <resource-file> is an initialisation file that tells the TextServer where to locate the resources that it needs, and PORT_NUMBER is the port on which the service is run. Infogistics' TextServer allows one to communicate with a number of text processing tools such as NLProcessor and Xtract (Infogistics' Information Extraction tool). Resources for all these tools can be specified in their own resource files and the resource file which is passed to the TextServer contains references to these individual resources.

Once the server is running, one can establish a connection to the TextServer and access its different functions through the IGTextClient program, which always takes a host name and port number as its arguments. To obtain diagnostic information, for instance, IGTextClient should be called with the argument –diagnose:

```
>> IGTextClient –host <HOST_NAME> -port PORT_NUM -diagnose
```

This program provides interface similar to the stand-alone individual tools which are encompassed by the TextServer. To call such a tool one need to specify the tool name, arguments for its operation, and data to work on. Here is an example of calling NLProcessor to tokenize an XML file:

```
>> IGTextClient –host localhost -port  2222 –tool nlproc –args '-qd ".*/DOC" –qs "."' file_to_process.xml
```

this call will add XML tokenization markup to the processed file, and output the result inot stdout.

The main reason to run the TextServer in this way, however, is that clients can exist on multiple deployment platforms which communicate with it, thus bringing its functionality to a wide variety of applications.

V. NLProcessor resources – fine tuning the performance

NLProcessor comes with some prebuilt resources such as grammars, lexicons, statistical models, etc. These resources are specified in the resource file resource.spc, which is located in the "Resource" directory of NLProcessor installation.

lists.cmp -- this lexicon contains a list of known abbreviations. If you notice that the system does not recognize abbreviations in your domain you can add these abbreviations to lists.cmp as follows

a) add abbreviations to data/RAW/lists.lex file under [abbreviation] category; if you want an entry to be case sensitive use *C flag. This file has format in which category in square brackets is followed by a list of words one per line which belong to this category. So to add new abbreviations you need to locate [abreviation] marker and add to existing abbreviations your new ones (without final period). For instance,

```
    [abbreviation]
    Abbr
    MyAbbr *C
```

specifies abbreviation Abbr which will be matched regardless of the case of its letters and abbreviation MyAbbr which will be matched in the case-sensitive way.

b) compile data/RAW/lists.lex to data/lists.cmp using comp_lex utility:

```
>> c:\NLP\bin\comp_lex -o \NLP\data\lists.cmp \NLP\data\RAW\lists.lex
```

itok_res.cmp - this file contains tokenization rules compiled into binary format. This file is compiled from the resource file data/RAW/itok_res.xml. The resource file contains the FSA in which every REX element specifies a regular expression, which should be matched to get a token of a certain type. For example:

```
<REX name=ORD>[0-9]+[\-]*((th)|(rd))</REX>
```

applies regular expression [0-9]+[\-]*((th)|(rd)) and if it is satisfied it assigns the tag ORD to the matched segment. The REX rules are applied in the same order as they are specified and the longest matching rule is selected. You can also specify a lookahead for a rule in the "bo" attribute of the REX element:

```
<REX name=ORD bo=1>[0-9]+[ \-]*((th)|(rd))[, ]</REX>
```

here we allowed for a whitespace in between e.g. "5 th" but to prevent incorrect tokenization (e.g. "<ORD>5 th</ORD>ousand") we also specified that this match should be followed by a comma or a whitespace and in order not to include it into the matched ordinal we specify the backoff of one position (i.e. bo=1).

After data/RAW/lttok_res.xml has been modified, it needs to be recompiled into data/lttok_res.cmp as follows:

```
>>c:\NLP\bin\comp_tok –o \NLP\data\itok_res.cmp \NLP\data\RAW\itok_res.xml
```

VI. NLProcessor SDK

NLProcessor can be integrated with your own applications as a dynamic library written in C. It comes with appropriate libraries: nlprocessor_win32.dll (Win32), nlprocessor_lnx.so (Linux) and nlprocessor_sol.so (Solaris). Compatible compilers: Microsoft Visual C++ 5.0 on WIN32 and higher, gcc 2.5 and higher on Linux, Solaris and SunOS.

To run an application linked with the NLProcessor dynamic library on a Win32 platform you will need to place nlprocessor_win32.dll to a directory specified in your PATH. To run an application linked with the NLProcessor dynamic library on a Linux or Solaris platform you will need to add the directory containing this library to your LD_LIBRARY_PATH environment variable.

To link your application to the NLProcessor dynamic library you will need to include nlprocessor_win32.lib, nlprocessor_lnx.so or nlprocessor_sol.so (depending on the platform) to the list of libraries you are linking with. All these libraries are MT safe and reentrant.

VI.1 Calling NLProcessor from your code

To be able to apply NLProcessor from your code you usually make the following sequence of API calls:

```
#include “nlprocessor.h”  //-- specifies NLProcessor API calls
.....................
const char* specs =”c:\NLP\resource.spc”;    //--  resource specification file
const char* args=NULL;        //-- session arguments (akin command line arguments)

XIN xin = nlprocessor_Init(specs, args);                  //- (1) init. and obtain handlefor the session
const char* text;
while(text= GET NEXT DOCUMENT TO PROCESS)
{
   int status = nlprocessor_Adapt2Text(xin,  text);      //--  (2) do  on-line pre-learning
   if(status!=nlprocessor_OK)                            //-- (3) check that processing was OK
      { fprintf(stderr, "%s", nlprocessor_Error(status); continue;} //-- (4) report
   status = nlprocessor_ProcessText(xin, text, args);   //-- (5) do processing
   if(status!=nlprocessor_OK)                            //-- (6) check that processing was OK
      { fprintf(stderr, "%s", nlprocessor_Error(status); continue;}
   char* xml_result = nlprocessor_GetXMLmarkup(xin, -1); //-- (8) get XML tokenized text
       DO SOMETHING USEFUL WITH xml_result AND FREE IT
   nlprocessor_ResetAdaptation(xin);                      //--(9) clean to be able to adapt to a new document
}
nlprocessor_Close(xin);   //-- (10) close NLProcessor session
```

First of all "nlprocessor.h" which specifies all the NLProcessor API calls has to be included. An NLProcessor session has to be initialized by calling the **nlprocessor_Init()** API call. This call returns a handle for the session, which is required for all other calls to NLProcessor functions. Multiple sessions can be opened at the same time. The nlprocessor_Init() function takes two strings as arguments: *specs* specifies the location of the resource, and *args* specifies command-line options.   *args* can be NULL, in which case the default settings will be used.

After a session has been initialized documents can be processing. In the "while" loop documents are read into the "text" variable one by one. Then the processor is adapted to this text by calling **nlprocessor_Adapt2Text()**. This function is called with the current NLProcessor session handle xin. During this call, the NLProcessor applies some on-line dynamic learning algorithms which allow it to produce more accurate results during processing. This call returns the status (error code) of its operation, which can be converted to a string by calling **nlprocessor_Error(status).**

Now the actual processing of the text can be performed by calling **nlprocessor_ProcessText()**. This function is called with the current NLProcessor session handle (xin) and the text itself. The *args* settings can also be supplied if we wish to change some settings from the ones given to the nlprocessor_Init() call. Otherwise *args* should be set to NULL. This call returns the status (error code) of its operation, which can be converted to a string by calling **nlprocessor_Error(status).**

There are numerous ways in which the results of tokenization can be accessed. Here we have used the **nlprocessor_GetXMLmarkup(xin, -1)** call which returns the entire XML marked up tokenization results into a newly allocated string. This function can return XML markup for individual sentences and tokens as well. To indicate that we wish the entire document to be returned we passed "-1" . More detailed description for accessing results of the processing is given in section VI.2.

After the text has been processed and tokenization results have been utilized, a new document can be processed. Before starting a new document it is necessary to reset the adaptation resources made by nlprocessor_Adapt2Text(). This is done by calling the **nlprocessor_ResetAdaptation(xin)** API function with current session handle xin, which resets the NLProcessor's internal state to its pre-adapted state.

To finish an NLProcessor session call **nlprocessor_Close(xin).**

VI.2 Obtaining the results of tokenization

There are numerous ways the results of tokenization can be accessed. Above we showed an example of using the **nlprocessor_GetXMLmarkup(xin, -1)** call which returns the entire text with XML marked up tokenization results into a newly allocated string. This function can return XML markup for individual sentences and tokens as well. The "-1" in the second argument is used to indicate that the entire text should be retrieved.

It is also possible to access the results of tokenization in a sentence by sentence or token by token fashion. In this mode first a handle for a sentence or a token needs to be obtained and then this sentence or token can be retrieved. To obtain a handle to a sentence the **nlprocessor_GetNextSentenceHandle(xin, prev_sentence_handle)** call should be used**,** and then we can, for instance, use nlprocessor_GetXMLmarkup() with the newly obtained sentence handle to get its XML representation:

```
 int sentence_handle=-1;     //-- set handler to –1 so its next element will be the starting one
   while((sentence_handle=
          nlprocessor_GetNextSentenceHandle(xin, sentence_handle)!=-1)
   {
     char* xml_sentence =
                         nlprocessor_GetXMLmarkup(xin, sentence_handle);
       DO SOMETHING USEFUL WITH xml_sentence AND FREE IT
   }
```

Instead of retrieving the XML representation of a sentence you can obtain handles to individual tokens by calling nlprocessor_GetNextTokenHandle(xin, prev_token_handle). After you have obtained a handle to a token you can retrieve its individual properties such as character content, delimiters (whitespace, etc.) to the left and to the right, token class, token span in the text. Of course, you can also obtain the XML representation of the token by calling nlprocessor_GetXMLmarkup() function. Here is an example how to iterate through tokens:

```
 int token_handle=-1; //-- set handle so its next element will be the starting one
while((token_handle=nlprocessor_GetNextTokenHandle(xin, token_handle)!=-1)
{
     char* xml_token = nlprocessor_GetXMLmarkup(xin, token_handle);
     DO SOMETHING USEFUL WITH xml_sentence AND FREE IT

     //-- returns character content of the token (word itself) -----
     char* body     = nlprocessor_GetTokenPropertyBody(xin, token_handle);
     //-- returns string which separates this token from the one to the left
     char* leftws   = nlprocessor_GetTokenPropertyDelimLeft(xin, token_handle);
     //-- returns string which separates this token from the one to the right
     char* rightws = nlprocessor_GetTokenPropertyDelimRight(xin, token_handle);
     //-- returns sentence handle this token belongs to
     int  sent_hdlr  = nlprocessor_GetTokenPropertySentence(xin, token_handle);
     //-- returns position (character number) this token starts from in the text
     int start_pos   = nlprocessor_GetTokenPropertyStart(xin, token_handle);
     //-- returns position (character number) this token ends at in the text
     int end_pos    = nlprocessor_GetTokenPropertyEnd(xin, token_handle);
     //-- returns token class (see section II. Token Output)
     char* tok_cls = nlprocessor_GetTokenPropertyClass(xin, token_handle);
     //-- returns flag: 0 or 1 whether token is capitalized in ambiguous position
     int amb_cap   = nlprocessor_GetTokenPropertyAmbCap(xin, token_handle);
}
```

Note, that all returned strings are not allocated but just pointers to internal fields of the current token and they will be lost after the next call to nlprocessor_Adapt2Text() or nlprocessor_ProcessText().

VI.3 Complete API Specification

XIN nlprocessor_Init(const char* specs, const char* args);
Initializes a session with the NLProcessor engine.
const char* specs - full path name for NLProcessor resource file;
const char* args   - arguments to override default (akin to options to standalone tool); can be NULL

returns XIN   - NLProcessor session identification handle

int nlprocessor_Adapt2Text(XIN xin, const char* text);
performs on-line learning from text.

XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
const char* text - text to perform adaptation to;

returns an error code or nlprocessor_OK if no error

int nlprocessor_ProcessText(XIN xin, const char* text, const char* args);
performs tokenization on text
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
const char* text - text to process;
const char* args - arguments to override default (akin to options to standalone tool); can be NULL

returns an error code or nlprocessor_OK if no error

const char* nlprocessor_Error(int status)
reports an error
int status -- error number (usually returned by nlprocessor_ProcessText()

returns a pointer to an internal error message

void nlprocessor_ResetAdaptation(XIN xin);
drops adaptation results and prepares The NLProcessor to be adopted to new text
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());

void nlprocessor_Close(xin);
ends an NLProcessor session and frees allocated resources
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());

char* nlprocessor_GetXMLmarkup(XIN xin, int obj_handle);
retrieves XML representation for an object of tokenization (text, sentence, token)
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int obj_handle – handle to the object of tokenization (text, sentence, token) to access
(-1 – get entire text otherwise this handle is usually obtained by
calling nlprocessor_GetNextTokenHandle() or nlprocessor_GetNextSentenceHandle()

returns a string containing the document, or a sentence or a token marked up by XML tags. This string is allocated.

int nlprocessor_GetNextTokenHandle(XIN xin, int prev_token_handle);
returns handle for next token
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int prev_token_handle – handle to the previous token (to get the first token, it must be set to –1)

returns the handle to a token which then can be used to access this token properties or XML representation. This handle can be then passed to the next call of nlprocessor_GetNextTokenHandle() as the second argument. If no token can be obtained for the given handle, this call returns –1.

int nlprocessor_GetNextSentenceHandle(XIN xin, int prev_sentence_handle);
returns handle for next sentence
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int prev_sentence_handle – handle to the previous sentence (to get the first sentence it must be set to –1)

returns the handle to a sentence which then can be used to access this sentence XML representation. This handle can be then passed to the next call of nlprocessor_GetNextSentenceHandle() as the second argument. If no sentence can be obtained for the given handle, this call returns –1.

char* nlprocessor_GetTokenPropertyBody(XIN xin, int token_handle);
returns character content of the token (word itself)
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a pointer to token body. It is statically allocated and will be lost with the next call to nlprocessor_Adapt2Text() or nlprocessor_ProcessText(). If no token can be obtained for the given handle, this call returns NULL;

char* nlprocessor_GetTokenPropertyDelimLeft(XIN xin, int oken_handle);
returns string of characters which separates this token from the one to the left
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a string of characters which separates this token from the one to the left. It is not allocated and will be lost with the next call to nlprocessor_Adapt2Text() or nlprocessor_ProcessText(). If no token can be obtained for the handle this call returns NULL;

char* nlprocessor_GetTokenPropertyDelimRight(XIN xin, int oken_handle);
returns string of characters which separates this token from the one to the right
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a string of characters which separates this token from the one to the right. It is not allocated and will be lost with the next call to nlprocessor_Adapt2Text() or nlprocessor_ProcessText(). If not token can be obtained for the handle this call returns NULL;

int nlprocessor_GetTokenPropertySentence(XIN xin, int token_handle);
returns sentence handle this token belongs to
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a sentence handle this token belongs to.; -1 if no token can be obtained for the handle.

int nlprocessor_GetTokenPropertyNG(XIN xin, int token_handle);
returns noun group handle this token belongs to
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a noun group handle this token belongs to.; -1 if no NG can be obtained for the handle.

int nlprocessor_GetTokenPropertyVG(XIN xin, int token_handle);
returns verb group handle this token belongs to
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a verb group handle this token belongs to.; -1 if no VG can be obtained for the handle.

int nlprocessor_GetTokenPropertyStart(XIN xin, int token_handle);
returns position (character number) this token starts from in the text
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a position (character number) this token starts from in the text; -1 if no token can be obtained for the handle.

int nlprocessor_GetTokenPropertyEnd(XIN xin, int token_handle);
returns position (character number) this token ends at in the text
XIN xin - NLProcessor session identification handle (obtained by nlprocessor_Init());
int token_handle - handle to a token to access (usually obtained by nlprocessor_GetNextTokenHandle());

returns a position (character number) this token end at in the text; -1 if no token can be obtained for the handle.

VII. Java API : package com.infogistics.nlprocessor

The Java API to NLProcessor is implemented through a CORBA-like access to the TextServer (IGTextServer) which is running as a service on a computer reachable from your target computer through TCP/IP (see section "NLProcessor – server mode" for more details). The Java API consists of following classes: **TSConnection**, **NLProcessor**, and xml traversing classes **Token**, **TokenIterator** and **Chunk**, which are all included in the **com.infogistics.nlprocessor** package. The main purpose of the TSConnection class is to create an NLProcessor object. The NLProcessor object applies processing to the text and returns a string which contains the original text with XML marked results of tokenization, part-of-speech tagging, sentence segmentation, etc. as explained in section "NLProcessor output". Since this result is a string containing valid XML it can be parsed by any XML parser e.g. XP. We also provide the classes Token, TokenIterator and Chunk which allow you to iterate over the different types of objects (words, parts-of-speech, sentences..) marked which the processor marks up in the text..

VII.1 class com.infogistics.nlprocessor.TSConnection (extends Object)

The TSConnection class manages all of the details of establishing the connection to the TextServer from your Java program. To access the TextServer on a single machine you need to construct one TSConnection object and release it after you have finished working with the TextServer. The main purpose of a TSConnection object is to construct NLProcessor objects which do the actual processing.

public TSConnection(String hostname, int port_nm) throws IOException

establishes a connection to the TextServer which is attached to the port specified in port_num on the server specified by hostname. After this, the TSConnection object is set to its "connected state", which can be checked by using isConnected() method. If the connection fail to be established, it throws an exception with a diagnostic string.

public void releaseConnection()

this method releases resources required for connection. It should be called before exiting your program or when you decide you do not need any further processing.

public static boolean isConnected()

Returns true if the object is succesfully connected to the TextServer and false otherwise

public Enumeration getAvailableProcessors() throws IOException, TSException

This call queries the TextServer to which a connection has been established and retrieves enumeration which contains strings which specify types of available processors. Throws an exception if the object is not connected to the TextServer.

public NLProcessor getNLProcessor(String descr) () throws IOException, TSException

This call creates an NLProcessor object of type specified by *descr* string. Throws an exception if the object is not connected to the TextServer or if *descr* is not identified as a valid processor key.

public boolean registerNLProcessor(String descr, String url_for_resource) throws IOException, TSException

This call loads a new NLProcessor into the TextServer, and initalizes it from the resource file specified by *url_for_resource* argument. This new NLProcessor is associated with the string *descr* as its key. This key then can be used by getNLProcessor(). Throws an exception if the object is not connected to the TextServer or rethrows exeptions produced during new object initalization.

A standard way to work with TSConnection class is:

```
TSConnection textServer=null;
NLProcessor defaultEnglishNLProcessor=null;
try {
   textServer = new TSConnection("localhost", 2222); //-- TextServer runs on port 2222
   Enumeration processors = textServer.getAvailableProcessors();
   while(processors.hasMoreElements())
   {
        String processor_type = (String) processors.nextElement();
        if(processor_type =="EDEFAULT")
        {
             defaultEnglishNLProcessor = textServer.getNLProcessor(hdl);
             break;
        }
   }
} catch (Exception e){System.err.println(e.toString()); System.exit();}

if(defaultEnglishNLProcessor==null)
{ System.err.println("No default English processor registered in the TextServer. Exiting\n");
   System.exit();
}

 //----- do something useful with defaultEnglishNLProcessor

textServer.releaseConnection();
```

Indeed, instead of iterating through the available servers we could use getNLProcessor("EDEFAULT") directly.

VII.2 class com.infogistics.nlprocessor.NLProcessor (extends Object)

The NLProcessor class performs the actual processing of text. The only way to construct an NLProcessor object is to call TSConnection.getNLProcessor() with a string which contains the processor type as its argument. Currently there is only one predefined processor "EDEFAULT" which is a default processor for English. New resources can be developed and can be realized using TSConnection .registerNLProcessor().

The NLProcessor class contains methods which allow text processing at the document level (processDocument()) and at the sub-document level (adaptToText() and processText()). There are several flags which can be set to control details of processing e.g. setDoTagging() switches on or off part-of-speech tagging functionality, setDoSentence() specifies whether identification of sentences in the text is needed, etc. The NLProcessor returns a string with original text marked up in XML as described above.

One of the smart characteristics of NLProcessor is its ability to learn certain regularities about text and then apply them during processing. We call this “self-adaptation”. When you process an entire document you don’t need to worry about controlling the self-adaptation functionality. When you process a document passage by passage there are three ways to control self-adaptation.

a) During the first pass run all text passages through self-adaptation process and only then run these passages through the actual processing. This is the most accurate way to process the text but at the same time it is most time-consuming.
b) Run the self-adaptation process on a text passage immediately prior to processing. In this case you don’t need to implement the double-pass control strategy as in the previous case, but the adaptation rules induced from the text passages cannot be applied to passages which precede it but only to the passages which follow it.
c) Not to use self-adaptation at all, and call only the process method. This may produce inferior results but is the fastest.

After you have finished processing passages from a single document it is necessary to reset the adaptation resources of the NLProcessor, because the regularities it has learned from the current document might not be consistent with your next document .

public void release( ) throws IOException

this method releases resources allocated in the TextServer for this instance of NLProcessor. It should be called before exiting your program or when you decide not to use this NLProcessor object. All calls to an object after release will result in throwing an exception with the message “NLProcessor has been released.”.

public String processDocument(String document) throws IOException, TSException

This call takes a string which contains the entire document and returns results of processing in a string which contains the original document with results of processing marked in XML. Throws an exception if there have been errors during processing or if object has been released.

public String processXMLDocument(String document, String query) throws IOException, TSException

This call takes a string which contains an XML document and returns the results of processing in a string, which contains the original document with results of processing marked in XML. The query attribute specifies label of XML elements within the XML document which character should be processed. If query is null all available character data will be processed. Throws an exception if there have been errors during processing or if object has been released.

public void adaptToText(String text) throws IOException, TSException

This call takes a string which contains some text and adapts the processor to it Throws an exception if there have been errors during processing or if object has been released.

public String processText(String text) throws IOException, TSException

This call takes a string which contains some text and returns the results of processing in a sting, which contains the original text with results of processing marked in XML. Throws an exception if there have been errors during processing or if object has been released.

public void resetAdaptation( ) throws IOException

This call resets the NLProcessor object to pre-adaptation stage. Throws an exception if object has been released.

public void setDoTagging(boolean b)

This call sets the do-tagging flag which specifies whether results of part-of-speech tagging should be provided with tokens.

public boolean isDoTagging( )

This call tests the value of the do-tagging flag.

public void setDoSentence(boolean b)

This call sets the do-sentence flag which specifies whether text should be segmented into sentences.

public boolean isDoSentence( )

This call tests the value of the do-sentence flag.

public void setDoTitle(boolean b)

This call sets the do-title flag which specifies whether proper titles such as “Phantom of the Opera” should be segmented in the text.

public boolean isDoTitle( )

This call tests the value of the do-title flag.

public void setDoNG(boolean b)

This call sets the do-ng flag which specifies whether text should be segmented into noun groups.

public boolean isDoNG( )

This call tests the value of the do-ng flag.

public void setDoVG(boolean b)

This call sets the do-vg flag which specifies whether text should be segmented into verb groups.

public boolean isDoVG( )

This call tests the value of the do-vg flag.

public boolean isReleased( )

This call returns true if the object has been relesed by release() method.

Here is a simple example of processing an ascii document by defaultEnglishNLProcessor which was created in section “class TSConnection”:

```
String result = defaultEnglishNLProcessor.processDocument(“This is a rainy day.”);
```

Here we apply procesing to an XML document:

```
String xml_document = “<DOC><P>This is a rainy day.</P> <P>I need a break.</P></DOC>”
String result = defaultEnglishNLProcessor.processXMLDocument(xml_document, “P”);
```

This call will process character data of P elements.
Here is an example how to process a document passage-by-passage applying self-adaptation strategy first. Here we use Enumeration which contains strings with document paragraphs which are obtained by getParagraphs() method from some object which contains a document:

```
Enumerator document = text.getPragraphs();  //-- some object text
defaultEnglishNLProcessor.setDoSentence(true); //-- segment sentences in the text
for(document.hasMoreElements()))
     defaultEnglishNLProcessor.adaptToText(document.nextElement()); //-- do adaptation

document = text.getPragraphs();
for(document.hasMoreElements()))
{
     String result = defaultEnglishNLProcessor.processText(document.nextElement());
    //----- do something useful with result  e.g. parse it with XML parser --------
}
defaultEnglishNLProcessor..resetAdaptation(); //-- prepare for a new document.

……………………………………
defaultEnglishNLProcessor.release();  //-- finish working with defaultEnglishNLProcessor
```

VII.3 Working with XML tokenization

Once text has been processed by the NLProcessor and went through the zoner, tokenizer, part-of-speech tagger and chunker, it is returned as a **String** object which contains an XML document. Although you are free to do with it as you wish, our API offers an abstract way to access the mark-up which has been added to the text. It consists from three classes Token, TokenIterator and Chunk.

A chunk is an aggregation of tokens and there are several types of chunks: the entire document, paragraphs, sentences, noun groups and verb groups. Tokens which constitute chunks apart from their character data can be queried for their type (word, number, etc.) and POS information. To iterate through tokens of a chunk there is class TokenIterator.

VII.3.1 class com.infogistics.nlprocessor.Token extends Object

This class encapsulates the tokens, possibly with their parts of speech, identified by the tagger. There are also empty tokens which correspond to starts and ends of token aggregations or chunks such as noun groups, verb groups, sentences and paragraphs.

Constants:

This class contains a number of constants which are used to represent type information about tokens.

Member	***Type***	***Description***
REG_WD	public static int	Regular word. Written lower-case in the middle of a sentence, and capitalized at the start of a sentence
NAME_WD	public static int	Proper name. The tokenizer thinks that this is a liekly proper name word which would be written with an initial capital letter wherever it appears.
REG_NUM	public static int	Number or numerical token. The token was identified as a number or numerical token.
ORD_NUM	public static int	Ordinals, such as 3rd, 12 th, etc..
PUNCT	public static int	A comma, quotation mark, or any non sentence-ending non alphanumeric character.
SENT	public static int	A period, question mark or exclamation mark.
EMAIL	public static int	An identified legal e-mail address.
DATE	public static int	Date expressions
URL	public static int	Web URLs.
TEL	public static int	Telephone and fax numbers.
CH_ST	public static int	Chunk start tag. This is an empty token i.e. with no character contents. This is returned for the start of chunks in the corpus. Chunks include *zones, paragraphs, sentences, noun groups and verb groups*.
CH_END	public static int	Chunk end tag. This is an empty token i.e. with no character contents. This is returned for the ends of chunks in the corpus.
WORD	public static int	This combines the two alpha categories REG_WD | NAME_WD. No single token can have this type, but it is a utility type for bit-masks.
TEXT	public static int	This combines all the regular tokens together, but does *not* include chunk start and end information. This is a utility type for bit-masks.

Constructors: no public constructors

Methods:

Return value	***Protoype***	***Description***
public	int getType( )	Returns the type of the token as bitmap of constants described above .
public	String getPOS( )	Returns the part of speech assigned to the token (if any). In the case of chunk start and end tokens (CH_ST and CH_END), this call returns the category of the entire chunk (e.g. "SENT", "EMB-SENT", “VG”, “NG” "PARA").
public	String	Returns the content of the token. For of chunk start and end tokens, this returns all character data which belongs to

	getContent( )	the chunk but no submarkup information.
public	String getSeparator( )	Returns any whitespace content *after* this token.
public	int getCharacterOffset( )	Returns the character offset of the start of the token from the position in its maximally enclosing chunk. This is useful for referring back to the original document. The returned value is the offset in the **String** that was *sent* to the tokenizer, not the **String** that was returned by it.

VII.3.2 class com.infogistics.nlprocessor.Chunk

A *Chunk* is a unit which contains tokens and/or other chunks.

Static constants:

The constants in this class represent the type of the chunk in question. The set of recognised chunk types is likely to increase as additional improvements are made to the zoners and parsers. Currently, the only recognised chunk types are *GLOBAL*, which is the entire extent of the chunk passed into the processor, *SENTENCE*, which is an inferred sentence, *PARAGRAPH*, which is an inferred paragraph, *VG*, which is a verb group and *NG* which is a noun group.

Constructors:

	Prototype	***Description***
public	Chunk(String xmlTokenizedText) throws MalformedChunkException	Generates a new chunk from an XML representation as a string. The argument **xmlTokenizedText** should be an XML element returned by the tokenizer.

Methods:

	Prototype	***Description***
public	TokenIterator getTokenIterator(int typeMask)	This returns the set of tokens within the chunk which match the type mask given in *typeMask*, considered as bit-field. The fields in the mask consist of the members of **TokenType**. The start and end markers of the current chunk are *not* in the TokenIterator returned.
public	TokenIterator getTokenIterator()	This returns the set of all tokens within the chunk.
public	Token getStartToken()	This returns the start token of a chunk. This empty token contains information about chunk type

VII.3.3 class com.infogistics.nlprocessor.TokenIterator implements Iterator

A *TokenIterator* is an iterator over tokens. It allows simple iteration through all the tokens identified by the tokenizer.

Constructors: no public constructors. Can be constructed only by calling method getTokenIterator() of the

Chunk class.

Methods:

	Prototype	***Description***
public	Boolean hasNext( )	Returns true if there are any more tokens, and false otherwise.
public	Object next( )	Returns the next token in the list. The returned object will always be of type **Token**.
public	Object next(Token t)	This method gives next element as normal next() method but of Token t is chunk start token it skips to the end of this chunk and gives first token which follows it.
public	synchronized void remove( )	Removes the token from the representation underlying the Chunk. Further calls to *getTokenIterator()* on the underlying **Chunk** will not return tokens which have been *removed*.

VII.3.4 Example

Here is an example of printing all real tokens (as opposed to structural start and end tokens) with their POS informations one per line:

```
public void printNames( String tokText )
{
        Chunk chunk=new Chunk( tokText );
        TokenIterator iter=chunk.getTokenIterator(Token.TEXT);
        while( iter.hasNext( ) )
       {
               Token t=(Token)iter.next( );
               System.out.println( t.getContent() + " " +  t.getPOS() + "\n");
        }
}
```