INFOGISTICS' XTRACTOR
[DRAFT VERSION]

October 2000

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

TABLE OF CONTENTS

I. XTractor – general description

I.1 Overview

XTractor by Infogistics is a successor to the Named Entity Recognition system from the University of Edinburgh which won the prestigious MUC-7 competition organized by the American Defence Research Agency (DARPA) in 1998.

XTractor is an Information Extraction engine which sifts through large volumes of texts and creates database records for the objects which are mentioned in the text, such as people, organizations, locations, vehicles, etc. Moreover, it normalizes and unifies information so, for example, if somebody is referred to as "John D. Smith", "John", "J. Smith", or even "he", XTractor can figure out that this is the same person. And most importantly, XTractor identifies the key relationships or links between the objects that it finds.

For example, from a sentence "White is a managing director of Cyber Corp." XTractor will identify that there is a person with surname "White", there is a company called "Cyber Corp." and it will create an "employment" link between these two objects. It will also record particular details of the employment: "managing director".

```
*PERSON id=1                                                  *ORGANIZATION id=2
   surname:     White                                             name:  Cyber Corp.
   occupation:  managing director       *XREF: employed _at #1 #2
```

XTractor can be linked to our document viewer where the objects found by XTractor are highlighted, and information aggregated from many different places of a document can be seen at a glance. For instance, at the beginning of a document XTractor identifies that John Smith is 42 year old, later on it appends that he works at Good Co. Ltd. and at the end of this document it identifies that he drives a red Nissan. XTractor makes sure that all this information is kept under a single record "John Smith", suitable for further processing by the computer, or for displaying to humans.

XTractor 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, Xtractor also comes with a toolkit for building resources for the extraction of customer-defined objects and relations.

XTractor 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.

I.2 XTractor – core engine

The core engine of XTractor supports identification of objects and their properties as follows:

Object	Properties
PERSON	first name, middle name, last name, title, age, DOB, occupation, passport number, personal id, social security number
LOCATION	housenum, housename, street, area, district, city, region, state, country, zipcode, geographical, full address;

ORGANIZATION	name, abbreviated name, alternative name
PRODUCT	name, manufacturer
TIME	seconds, minutes, hours, timezone
DATE	weekday, monthday, month, year, century, holiday
CONTACT	telephone, fax, email
VEHICLE	make, model, year, registration
MONEY	amount, currency
MEASURE (weight, distance, duration, percent, etc.)	– type, amount, units
RELATION	relative (father_of, mother_of, brother_of…), homeaddress, businessaddress, born_at, empoyed_at, contact_on, etc.

We have also successfully developed additional resources for identifying objects that are specific to the medical domain (drugs, medical conditions, chemical formulae..), property market (types of properties, specifications, areas…), legal documents, and some other more specialised domains.

II XTractor output

XTractor can deliver its output in two modes: extraction mode and markup mode.

II.1 XTractor output – extraction mode

In the *extraction mode* XTractor processes text and outputs object records. An object record encodes object type (e.g. person, location, etc.), properties of this object (e.g. surname for person, model for vehicle, etc.) and locations where it was extracted from in the text. Such records are marked using an XML markup with the structure as follows:

```
<object-type src="xtractor" cls="obj" id=object-id val=NormalizedString >
   <span sl=start-line sc=start-char el=end-line ec=end-char>Text</span>
   ……………………………………………………………………………
   <span sl=start-line sc=start-char el=end-line ec=end-char>Text</span>

   <property-type cls="property" val=NormalizedAttributeValue>
      <span sl=start-line sc=start-char el=end-line ec=end-char>Text</span>
       ………………………………………………………………………
     <span sl=start-line sc=start-char el=end-line ec=end-char>Text</span>
   </property-type>
```

```
  ……………………………………………………………
  <property-type  cls="property"  val=NormalizedAttributeValue>
     <span  sl=start-line  sc=start-char  el=end-line  ec=end-char>Text</span>
      ………………………………………………………………………
    <span  sl=start-line  sc=start-char  el=end-line  ec=end-char>Text</span>
   </property-type>
</object-type>
```

An object record represents an object of a certain type e.g. person, organization, location, etc. This type is specified in the label (object-type) of an object record. An object record has two fixed attributes: attribute src="*xtractor*" indicates that this record was produced by XTractor and attribute src="*obj*" indicates that this is an extracted object. These attributes can be used to automate conversion of a set of object records into your own format (or to SQL statements) – you don't need to list all possible object-types since the information provided by these two attributes indicates unambiguously that this is an object record. The val attribute keeps a normalized string representation of this object but it can be empty if the normalized string is identical to the entire character body of the object. Each object record has a unique id which is specified in the id attribute.

The information about the exact locations where from in the text this object was extracted is provided in the span elements which contain start line (sl), start character (sc), end line (el) and end character (ec) information together with the exact string in this span.

The individual properties of an object are encoded in the property elements. The actual list of properties is dependent on the type of the object. The structure of a property element is similar to that of the object: a property element can have its normalized value and a list of span elements, which provide the information on where this property was extracted from. Like object elements, property elements specify their type (surname, model, etc.) in the label and have a fixed attribute cls, which in this case set to be *property* rather than *obj*.

Here is an example of an object record. This record aggregates all mentionings of a person called Robert Johns in the text:

```
<person src="xtractor"  cls="obj"  id="1"  val="Johns, Robert" >
  <span sl="2"   sc="23"  el="2"   ec="36">Robert Johns</span>
  <span sl="5"   sc="3"    el="5"   ec="12">R. Johns</span>
  <span sl="19" sc="44"  el="19" ec="49">Johns</span>

  <firstname cls="property" val="Robert">
     <span sl="2" sc="23"  el="2"  ec="29">Robert</span>
     <span sl="5" sc="3"    el="5"  ec="8">R.</span>
   </firstname>

  <surname cls="property" val="Johns">
    <span sl="2"   sc="31"  el="2"   ec="36">Johns</span>
    <span sl="5"   sc="7"    el="5"   ec="12">Johns</span>
    <span sl="19" sc="44"  el="19" ec="49">Johns</span>
  </surname>
</person>
```

First we see that this record is of type person. For person objects the normalization standard currently is "Surname, Firstname" and the normalized string "Johns, Robert" is kept in the val attribute. The span elements show that this person is mentioned three times in the text: first time as "Robert Johns", then as "R. Johns" and finally simply as "Johns". The exact locations of these occurrences are also provided. For instance, "R. Johns" is mentioned on the line 5 starting at the 3rd character and ending at the 12th character.

Under the main object the individual properties for this object are listed. There are two of them: firstname and surname. Each of these properties specify its normalized value in the val attribute. For instance, the first name has two mentionings: "Robert" and "R." but the normalized (main) one is Robert. The span elements under properties specify where from in the text they have been extracted.

II.2 XTractor output – markup mode

In the *markup mode* XTractor tags the text with XML markers, which signal start and end of found objects and their properties. The XML encoding is similar to that employed by the *extract mode* but it does not use the span elements since the locations are annotated in the text directly. In this mode we also employ one extra attribute (idref) for object elements. This attribute specifies that marked at a certian location object has already been identified earlier in the text and the value of the idref attribute specifies the id of it. Thus one can assemble aggregated object records as produced in the *extract mode*. For example, the sentence

Robert Johns is a gardener.

will be marked up as

```
<person src="xtractor" cls="obj" id="1" val="Johns, Robert">
   <firstname cls="property">Robert</firstname>
   <surname  cls="property">Johns</surname>
</person>
 is a
<person src=xtractor cls="obj" id="2" idref="1">
    <occupation cls="property">gardener</occupation>
</person>.
```

In this text XTractor found two objects of type person: "Robert Johns" and "gardener". These objects are explicitly marked in the text. The fact that these two objects refer to a single person is represented by the idref attribute of the second object: this idref attribute is set to the id (1) of the object it should be unified with.

III. XTractor – standalone tool

The standalone version of XTractor works both with plain ASCII texts and with texts marked up in XML. The standalone version can also be used in server mode, which communicates with a calling program via system sockets to operate as a service.

To call XTractor 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 XTractor 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 XTractor which uses the standard resources to process text from the file "file_to_process", which is an ASCII file. We assume that your XTractor installation is in C:\XTRACTOR on a Win32 platform:

```
>>xtract   c:\XTRACTOR\Resource\resource.spc  file_to_process
```

The default output mode of xtract is *extract.* To switch to the markup mode use –markup option:

```
>>xtract   -markup c:\XTRACTOR\Resource\resource.spc file_to_process
```

When you run XTractor 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:

```
  >>xtract   -qd ".*/DOC" –qs  ".*/P" c:\XTRACTOR\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 you –q "." which means get text under parent element which in this case is 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 LTXML Query Language document.

To switch to the markup mode use –markup option:

```
>>xtract   -markup -qd ".*/DOC" –qs  ".*/P" c:\XTRACTOR\Resource\resource.spc  file_to_process.xml
```

Other command-line options:

-id NUM   -- start id counts for extracted objects with NUM (default 0);

-id_speaker NUM – XTractor automatically creates speaker object and you might wish to assign this object a specific id rather than let XTractor to generate one. This is useful when you want to unify this processing with processing of a different document of the same Author.

Here is an example how you can make the XTractor to build objects with ids starting form 10:

```
>>xtract   -markup  -id 10 c:\XTRACTOR\Resource\resource.spc  file_to_process
```

IV. XTractor – server mode

In addition to the stand-alone tools and SDK API, XTractor can also be delivered as a service within Infogistics' TextServer. As a module of Infogistics' TextServer, XTractor can easily be integrated in a distributed environment, and is particularly suited to being used with Enterprise Java Beans and COM objects. Since all data communication between TextServer and the Java application is in XML, Infogistics' XML Java wrappers can be used to access the linguistic structure that iToken adds to the document.

TextServer is architected as a TCP/IP-based service which communicates with an application via an efficient protocol. It is packaged with EJB (J2EE) session beans which interface with it, making this the ideal solution for integrating all of Infogistics' text analysis products into your Java application. It also comes with a COM interface which again integrates Infogistics' technology into this popular Microsoft platform.

We now describe the iToken functionality of the Java interface to TextServer.

IV.1 Running XTractor in TextServer

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

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

For example,

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

where resource-file is an initialisation file that tells 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 iToken (the tokenizer), tTag (Infogistics' POS tagger), tChunk (Infogistics' noun group chunker) 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 running, one can establish a connection to the TextServer and access its different functions through IGTextClient calling program which always takes host name and port number as its options. To obtain diagnostics information, for instance, IGTextClient should be called with the option –diagnose:

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

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

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

this call will output extracted objects marked up in XML (extract mode is default) for the processed file.

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

VI.1 TextServer JavaBean

Once TextServer is running, it is easy to interface with it using Java. TextServer comes bundled with some *Enterprise Java Beans* which allow a J2EE application to offer String-based, file-based and XML-based linguistic processing capabilities.

TextServer's Java interface consists of the following Session and Entity beans for iToken functionality.

- [TServer]: This bean encapsulates information about the TextServer, generates new TSession beans, and provides information about the status of the TextServer.
- [Tsession]: This session bean opens a communication session with the TextServer, and implements all of the interface functionality that iToken provides. Using this bean, it is possible to use iToken both in plain text and in XML mode, as well as to use other TextServer functionality you have opted to include.
- [TDocument]: This session bean represents an individual tokenized document. It gives a high-level interface to the tokenized representation of the document as processed by the TextServer.

The interfaces which these beans support essentially replicate the functionality of the SDK C interface described at the end of this document, and therefore is not included here. It facilitates the processing of documents and text in all of the formats supported by iToken.

V. XTractor resources – fine tuning the performance

For its operation XTractor comes with prebuilt resources such as grammars, lexicons, statistical models, etc. These resources are specified in the resource file resource.spc which is in the main XTractor directory. The standard resource file has structure as follows:

```
grammar-file          c:\XTRACTOR \resource\coregram.cgr        #-- grammar file for core engine
standard-lex-file     c:\XTRACTOR \resource\corelex.cmp         #-- lexicon file for core engine
user-lex-file         c:\XTRACTOR \resource\user.lex            #-- file in which you can add your
                                                                #-- own entries to the lexicon
system-lex-file       c:\XTRACTOR \resource\syslex.cmp          #-- support lexicon
preference-file       c:\XTRACTOR \resource\preference.lst      #--  session control preferences
```

You can fine-tune XTractor to a specific task by manipulating user.lex and preference.lst files. You can put strings with their types that you want XTractor to recognize into **user.lex**. You can add to the following categories:

firstname, surname,
street, town, state, geographical
organization-name
vehicle-make, vehicle-model

for example,

```
svetobor :: firstname
zhuguli   :: vehicle-make
moscow  :: town
```

Strings you add to the lexicon are case insensitive, but if you want it to be case sensitive you need to specify *C flag:

```
White :: lastname *C
```

The **preference.lst** file specifies a list of flags, which can switch on or off geographical localization.
The following table shows which countries currently have localisation instructions in place for them. In most cases, localisation is also possible at the state (province, etc.) level. It's important to remember, when localizing for a particular state or province (e.g. Ontario), to also localize for the country itself (i.e. Canada).

Country	State-level?	Format
US	Y	USXX, where XX is 2 letter state code, e.g. USFL
UK	Y	name of county, e.g. Lothian
Canada	Y	CAXX, where XX is 2 letter province code, e.g. CAON
Australia	Y	AUSXX(X), where XX(X) is 2 (or 3) letter state code, e.g. AUSNSW
Mexico	N	

The localization flags are activated by the "activate" command, e.g. to activate localization for Florida you do

```
activate USFL          #-- activate Florida location handling
```

to comment a line in the preference file use #.

VI. XTractor SDK

XTractor is available for the integration with your own applications as a dynamic library written in C: xtractor_win32.dll (Win32), xtractor_lnx.so (Linux) and xtractor_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 your application linked with the XTractor dynamic library on Win32 you need to place xtractor_win32.dll to a directory specified in your PATH. To run your application linked with the XTractor dynamic library on Linux or Solaris platform you need to add the directory where this library is contained to your LD_LIBRARY_PATH.

To link your application to XTractor dynamic library you need to include xtractor_win32.lib, xtractor_lnx.so or xtractor_sol.so (depending on the platform) to the list of libraries you are linking with.

VI.1 Calling XTractor from your code

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

```
#include "xtractor.h"  //-- specifies XTractor API calls
…………………
const char* specs =”c:\XTRACTOR\resource.spc”;   //--   resource specification  file
const char* prefs =NULL;     //-- preferences specification (will be describe later)
const char* args=NULL;     //--   session arguments (akin command line arguments) (will be describe later)

XIN xin = xtractor_Init(specs, prefs, args);       //---  (1)  initalize and obtain  handle xin for  the session
const char* text;
while(text= GET NEXT DOCUMENT TO PROCESS)
{
      int status = xtractor_Adapt2Text(xin,  text);        //--  (2) do tuning of XTractor to this document
      if(status!=xtractor_OK)                                    //-- (3) check that processing was OK
           { fprintf(stderr, “%s”, xtractor_Error(status); continue;} //-- (4) if not report error
      status = xtractor_ProcessText(xin, text, prefs, args);           //-- (5) do processing
      if(status!=itoken_OK)                                       //-- (6) check that processing was OK
           { fprintf(stderr, “%s”, xtractor_Error(status); continue;} //-- (7) if not report error
      char* xml_result = xtractor_GetXMLmarkup(xin);     //-- (8) get XML tokenized text
      DO SOMETHING USEFUL WITH xml_result AND FREE IT

      xtractor_ResetAdaptation(xin);    //--  (9) clean adaptation to be able to adapt to a new document
}
xtractor_Close(xin);  //-- (10) close XTractor session
```

First of all you need to include xtractor.h file which specifies all XTractor API calls. Then you need to initalize an XTractor session by calling **xtractor_Init()** API call. This call returns a handle for the session, which you will require for all other calls to XTractor functions. You can open multiple sessions at the same time. The xtractor_Init() function takes three strings as arguments: *specs* specifies the location of the resource, *prefs* specifies preferences for this session (use US time format, apply geographical lists specific to Florida, etc.), and *args* specifies the output mode, start id number, etc. Both *prefs* and *args* can be NULL in which case the default settings will take place. The exact content of *prefs* and *args* will be described later in greater detail.

After you have initialized a session you can send a document for processing. In the "while" loop we read one document after another into the "text" variable. Then we adapt XTractor to this text by calling **xtractor_Adapt2Text()**. We call this function with our current XTractor session handle xin and send it our current text. During this call XTractor applies some on-line dynamic learning algorithms which allow it to produce more accurate results during processing. This step however is not obligatory and if processing time is crucial you might opt for commenting it out. This call return status (error code) of its operation, which can be printed out by **calling itoken_Error(status).**

Now we are ready to perform actual processing of the text by calling **xtractor_ProcessText()**. We call this function with our current XTractor session handle (xin) and send it the text to process. We also might specify prefs and args settings if we wish to change some parameters from the ones set at the xtractor_Init() call, otherwise we may simply set prefs and args to NULL. This call return status (error code) of its operation, which can be printed out by **calling itoken_Error(status).**

There are several ways you can access the results of tokenization. Here we demonstrated **xtract_GetXMLmarkup(xin)** call which returns the entire text with XML marked up objects and their properties (see markup mode) into a string which it allocates. More detailed description for accessing results of the processing is given in section VI.3.

After you have processed or saved the extraction results, you can start working on the next document. But before you can do that, you should reset the adaptation resources produced by xtractor_Adapt2Text() if you want Xtractor to adapt itself to the next document (by calling xtractor_Adapt2Text()). This can be achieved by calling the **xtractor_ResetAdaptation(xin)** API function with current session handle xin, which resets Xtractor's internal state to its pre-adapted state.

To finish an XTractor session call **xtractor_Close(xin).**

VI.2 Obtaining the results of tokenization

There are several ways you can access the results of tokenization. Above we demonstrated **xtract_GetXMLmarkup(xin)** call which returns the entire text with XML marked up objects and their properties (see markup mode) into a string which it allocates**.** There is also a call **xtract_GetXMLextract(xin)** which in a similar way allocates a returned string and puts there the extracted objects with their properties in XML format (see extract mode).

There is also a way of accessing extracted objects and their properties in a sequential fashion. The structure of objects and properties is described in section "Xtractor output – extraction mode": each object has a type, a normalized value and a set of span elements which record positions in the text this object was extracted from. An object also contains a set of properties which have a similar structure.

In the sequential access mode we first need to obtain a handler for an object and then we can retrieve its individual properties and span information. To obtain a handle to an object we can call **xtractor_GetNextObjectHandler().** After we have obtained a handler for an object we can retrieve its type by calling **xtractor_GetType()** and its normalized value by calling **xtractor_GetVal()**.

We can then iterate through the span elements of the object by calling **xtractor_GetNextSpanHandler ()** and then obtain values of starting and ending positions and textual contents for each individual span. In a similar fashion we can iterate through individual properties of the object and then retrieve type and normalized value for a property as well as iterate through span elements of a property. Here is a sample code which does this:

```
int object_handler=-1;   //-- set handler so its next element will be the starting one
while((object_handler=xtractor_GetNextObjectHandler(xin, object_handler)!=-1)
{
    const char* obj_type = xtractor_GetType(xin, object_handler); //-- get object type e.g. PERSON
    const char* obj_val   = xtractor_GetVal(xin, object_handler);   //-- get norm. value: e.g. Johns, Robert

    int obj_span_handler=-1;   //-- set handler so its next element will be the starting one
    while((obj_span_handler=xtractor_GetNextSpanHandler(xin, object_handler, obj_span_handler)!=-1)
    {
        const char* span_txt = xtractor_GetSpanText(xin, obj_span_handler);
        int start_line = xtractor_GetSpanStartLine(xin, obj_span_handler);
        int start_pos = xtractor_GetSpanStartPos(xin, obj_span_handler);
        int end_line = xtractor_GetSpanEndLine(xin, obj_span_handler);
        int end_pos = xtractor_GetSpanEndPos(xin, obj_span_handler);
    } //-- end of obj_span_handler

    int prop_handler=-1;     //-- set handler so its next element will be the starting one
    while((prop_handler=xtrcator_GetNextPropertyHandler(xin, object_handler, prop_handler)!=-1)
    {
       const char* prop_type = xtractor_GetType(xin, prop_handler); //-- get property type e.g. LASTNAME
       const char* prop_val   = xtractor_GetVal(xin, prop_handler);  //-- get property norm. value e.g. Robert

        int prop_span_handler=-1;    //-- set handler so its next element will be the starting one
        while((prop_span_handler=xtracor_GetNextSpanHandler(xin, prop_handler, prop_span_handler)!=-1)
        {
            const char* span_txt = xtractor_GetSpanText(xin, prop_span_handler);
            int start_line = xtractor_GetSpanStartLine(xin, prop_span_handler);
            int start_pos = xtractor_GetSpanStartPos(xin, prop_span_handler);
            int end_line = xtractor_GetSpanEndLine(xin, prop_span_handler);
            int end_pos = xtractor_GetSpanEndPos(xin, prop_span_handler);
       } //-- end of prop_span_handler loop
     } //-- end of prop_handler loop
} //-- end of object_handler loop
```

Note that calls to **xtractor_GetNextSpanHandler()** take as their second argument a handler to an object or a property where this span belongs to. Calls to **xtrcator_GetNextPropertyHandler()** take as their second argument a handler to an object a property belongs to. Calls which retrieve information for types and normalized values of individual objects and their properties **xtractor_GetType()** and **xtractor_GetVal()** as well as **xtractor_GetSpanText()** which retrieves textula content of an individual span element return only pointers to the strings. These strings will be destroyed with the next call to xtractor_Adapt2Text() or xtractor_ProcessText().

VI.3 Specification of Preferences

As was explained in Section "XTractor resources – fine tuning the performance" there are two files, user.lex and preference.lst, which can be modified by the user to fine-tune the performance of XTractor. The same information can be passed to XTractor during session initialization through the **xtractor_Init()** API call in its second argument (prefs). You may specify lexical entries and flags exactly as you would specify them in the corresponding files e.g.:

```
activate USFL        #-- activate Florida location handling
```

```
svetobor :: firstname      #-- add new firstname

zhuguli   :: vehicle-make  #-- add Russian car maker to the list of vehicle makes

White :: lastname *C    #-- add White as potential surname but only if it starts with a capital letter
```

The information specified through the second argument of **xtractor_Init()** API call takes precedence over information which comes from user.lex and preference.lst files.

You can also specify session arguments which override the default values. The default output mode for XTractor is "extract". To change it to the "markup" mode you may use the "–markup" option on the command line when working with the standalone tool, and similarly can use the "–markup" option in the args string as well. You can also specify the starting id as "–id NUM". Indeed, you can similarly specify any command line option in the args string. At the moment these are:

-markup –id NUM –id_speaker NUM

You can dynamically update both the prefs and the args values simply by passing them to the **xtractor_ProcessText()** API call. If you do not wish to update these values you may pass NULL instead, which will cause XTractor to use the values for these parameters which were passed to **xtractor_Init**.

VI.4 Complete API Specification

XIN xtractor_Init(const char* specs, const char* prefs, const char* args);
Initializes a session with Xtractor engine.
const char* specs - full path name for Xtractor resource file;
const char* prefs - preferences to override default (see "Specification of Preferences"); can be NULL
const char* args - arguments to override default (see "Specification of Preferences"); can be NULL

returns XIN - XTractor session identification number

int xtractor_Adapt2Text(XIN xin, const char* text);
performs on-line learning from text.
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
const char* text - text to perform adaptation to;

returns error code or xtractor_OK if no error

int xtractor_ProcessText(XIN xin, const char* text, const char* prefs, const char* args);
performs extractiron on text
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
const char* text - text to process;
const char* prefs - preferences to override default (see "Specification of Preferences"); can be NULL
const char* args - arguments to override default (see "Specification of Preferences"); can be NULL

returns error code or xtractor_OK if no error

returns: string with results of extraction. This is either a set of extracted objects (extract mode) or the original text marked up with XML tags (markup mode). This string is allocated.

void xtractor_ResetAdaptation(XIN xin);
drops adaptation results and prepares Xtractor to be adopted to new text

XIN xin - XTractor session identification number (obtained by xtractor_Init() call);

void xtractor_Close(xin);
ends an Xtractor session and frees aloocated resources
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);

char* xtract_GetXMLmarkup(XIN xin);
retrieves the entire text marked with XML tags (see "markup mode" for exact format);
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);

returns string in which the XML marked up text is placed. This string is allocated so you will eventually need to free it.

char* xtract_GetXMLextract(XIN xin);
retrieves extracted objects marked with XML tags (see "extract mode" for exact format);
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);

returns string in which the XML marked up text is placed. This string is allocated so you will eventually need to free it.

int xtractor_GetNextObjectHandler(XIN xin, int prev_object_handler);
retrieves handler to an extracted objetc
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int prev_object_handler – handler to the previous object (to get the first object it must be set to –1)

returns handler to an object which then can be used to access this object properties or span elements. This handler can be then passed to the next call of itoken_GetNextObjectHandler() as the second argument. If no object can be obtained for the handler this call returns –1.

int xtrcator_GetNextPropertyHandler(XIN xin, int object_handler, int prev_prop_handler);
retrieves handler to a property of an object
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int object_handler – handler to the object this property should belong to
int prev_prop_handler – handler to the previous property (to get the first object it must be set to –1)

returns handler to a property which then can be used to access attributes of this properties or span elements. This handler can be then passed to the next call of itoken_GetNextPropertyHandler() as the third argument. If no property can be obtained for the handler this call returns –1.

const char* xtractor_GetType(XIN xin, int handler);
retrieves type of an object or a property
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int handler - handler to an object or a property

returns a pointer to a string which contains type name. . It is not allocated and will be lost with the next call to xtractor_Adapt2Text() or xtractor_ProcessText(). If no object or property can be obtained for the handler this call returns NULL;

const char* xtractor_GetVal(XIN xin, int handler);
retrieves normalized value for an object or a property
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int handler - handler to an object or a property

returns a pointer to a string which contains normalized value. It is not allocated and will be lost with the next call to xtractor_Adapt2Text() or xtractor_ProcessText(). If no object or property can be obtained for the handler this call returns NULL;

int xtractor_GetNextSpanHandler(XIN xin, int parent_handler, int prev_span_handler);
retrieves   handler to a span element of a object or a property
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int parent_handler – handler to the object or a property this span should belong to
int prev_span_handler – handler to the previous span (to get the first object it must be set to –1)

returns handler to a span which then can be used to access attributes of this span. This handler can be then passed to the next call of itoken_GetNextSpanHandler() as the third argument. If no span can be obtained for the handler this call returns –1.

const char* xtractor_GetSpanText(XIN xin, int span_handler);
retrieves   textual value of the span
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int span handler - handler to the span

returns a pointer to a string which contains textual value of the span. It is not allocated and will be lost with the next call to xtractor_Adapt2Text() or xtractor_ProcessText(). If no span element can be obtained for the handler this call returns NULL;

int xtractor_GetSpanStartLine(XIN xin, int span_handler);
retrieves line number the span starts with
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int span handler - handler to the span

returns line number the span starts with. If no span element can be obtained for the handler this call returns -1;

int xtractor_GetSpanStartPos(XIN xin, int span_handler);
retrieves position number in the line where the span starts
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int span handler - handler to the span

returns position number in the line where the span starts. If no span element can be obtained for the handler this call returns -1;

int xtractor_GetSpanEndLine(XIN xin, int span_handler);
retrieves line number the span ends at
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int span handler - handler to the span

returns line number the span ends at. If no span element can be obtained for the handler this call returns -1;

int xtractor_GetSpanEndPos(XIN xin, int span_handler);
retrieves position number in the line where the span ends
XIN xin - XTractor session identification number (obtained by xtractor_Init() call);
int span handler - handler to the span

returns position number in the line where the span ends. If no span element can be obtained for the handler this call returns -1;