Xerox Incremental Parser

XIP API

User's Guide

xerox

XEROX RESEARCH CENTRE EUROPE

6 CHEMIN DE MAUPERTUIS
38240 MEYLAN
FRANCE

© Copyright 2003 2014 Xerox Corporation. All rights reserved.

Copyright protection claimed includes all forms and matters of copyrightable material and information now allowed by statutory or judicial law or hereinafter granted, including without limitation, material generated from the software programs which are displayed on the screen such as icons, screen displays, looks, etc.

XIP ®, Xerox ®, The Document Company®, and all Xerox products mentioned in this publication are trademarks of Xerox Corporation.

Table of Contents

CONTENTS **3**

ABOUT THIS MANUAL **5**

CHAPTER 1: OVERVIEW **6**

XIP Architecture **6**
- About the Input Data 7
- About the Core XIP Linguistic Units 8
- About the Input Control Rules 8
- About Disambiguation 8
- About Chunking 9
- About Building Dependencies 9

Introduction to a XIP Grammar **10**

CHAPTER 2: LAUNCHING XIP USING THE COMMAND LINE **11**

Displaying XIP Results **12**

CHAPTER 3: THE XIP API – REFERENCE GUIDE **14**

Introduction to the API **14**
- Encoding Features 14
- Encoding Categories 14

Initializing and Managing a XIP Grammar **15**
- Loading a Grammar 15
- Loading a Parameter File 16
- Testing the Existence of a Grammar 16
- Freeing a Grammar from Memory 16

Parsing Information **17**
- Parsing Files 17
- Parsing Strings 18

Returning Results **19**
- Storing Results Using the VECTA Template 19
- Using the XipResult Class 20
- Using the XipUnit Class 20
- Using the XipNode Class 21
- Using the XipDependency Class 22
- Using the XipFeature Object 22

System Management Functions **23**
- Managing XIP Libraries 23
- Verifying the XIP License 23

API Basic Example **23**

APPENDIX A: THE XIP API - INCLUDE FILE **25**

APPENDIX B: THE XIP API - EXAMPLE 31

APPENDIX C: THE XIP XML DTD.....36
- Defining XIP DTD Elements.....36
- Defining XIP DTD Attributes 37
- Example XIP DTD 40

GLOSSARY OF TERMS.....41

About This Manual

The Xerox Incremental Parser (XIP) is a parser that takes textual input and provides linguistic information about it. XIP can modify and enrich lexical entries, construct chunks and other types of groupings, and build dependency relationships.

This manual introduces the major concepts and notions of how to use XIP grammars. This manual is intended for users new to XIP and therefore does not describe all of the XIP functionalities. See the *XIP Reference Guide* for complete details on XIP grammar development and XIP parser.

Audience

This manual is designed for linguists and computational linguists who are well acquainted with linguistic terminology and strategy.

Conventions Used

This manual uses the following style conventions:

- Monospaced font: this typeface is used for any text that appears on the computer screen or text that you should type. It is also used for file names, functions, and examples.
- *Monospaced italic font*: this typeface is used for any text that serves as a placeholder for a variable. For example, *layer* indicates that the word *layer* is a placeholder for an actual layer number, such as 15.
- Internal cross-references: this format is used to indicate cross-references within the manual. If you are working on an electronic copy of the manual, click the cross-reference to go directly to the section it references.

What This Manual Contains

This manual contains the following sections:

- Overview:
  Introduces how XIP parses text, including a look at the architecture of XIP and an overview of each XIP module.
- Implementing a XIP Grammar:
  Provides information on how to configure, initialize, and launch XIP.

Other XIP Documentation

In addition to this manual, the documentation set for XIP includes the *XIP Reference Guide*, which provides reference information about writing XIP grammars.

Chapter 1: **Overview**

This chapter provides an overview of how XIP parses input text. It includes the following sections:

- XIP Architecture
- Introduction to a XIP Grammar

XIP Architecture

XIP takes as input plain or processed text, which it transforms and analyzes. XIP can disambiguate lexical input, segment sequences of linguistic units, and extract dependencies. A dependency is a linguistic relationship between linguistic units.

XIP is composed of three core modules that are included with XIP by default and two optional pre-processing modules. The core modules are:

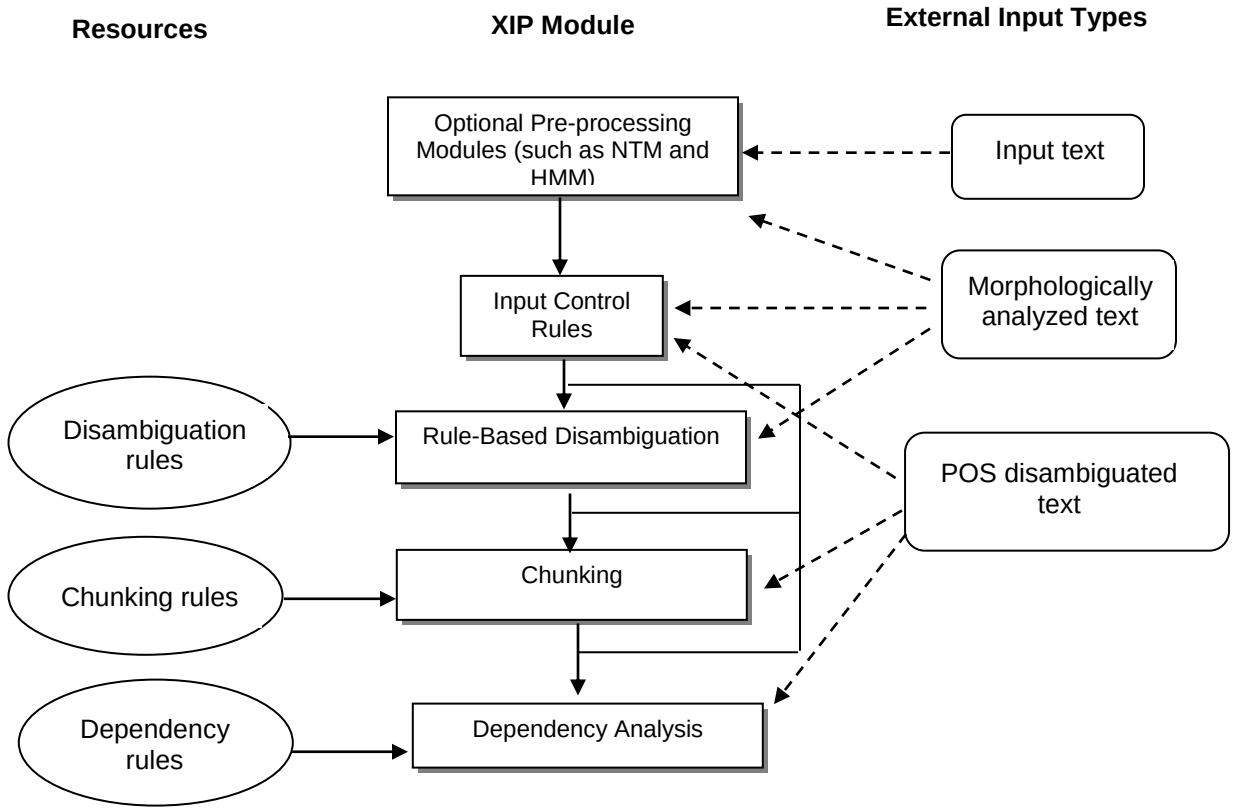
- Contextual disambiguation module: this module uses XIP rules to assign words features or categories according to their context.
- Chunking module: this module segments linguistic units as a series of chunks or other groupings.
- Dependency module: this module uses rules to identify dependencies between linguistic units.

The optional pre-processing modules are:

- Normalization, Tokenization, and Morphology (NTM): this module provides the normalized form and all potential lexical information for each word identified.
- Hidden Markov Model (HMM) disambiguator: this module uses the Hidden Markov Model algorithm to find the most probable grammatical category of a word according to its immediate context.

Different modules are activated depending upon the input data. The following diagram illustrates how the different modules interact in the XIP system:

Figure 1: Architecture of XIP



The remainder of this section describes the input data accepted by XIP and each phase of the XIP analysis of linguistic input.

About the Input Data

As illustrated in the Figure 1, XIP can take different kinds of linguistic objects as input, including:

- Raw ASCII text.
- A sequence of tokenized and morphologically analyzed words.
- A sequence of disambiguated words.
- A sequence of constituent structures such as NP, PP, and VP, which are conformant with the XIP XML DTD. For more information about the XIP XML DTD, refer to Appendix B in the *XIPReference Guide*.
- XML input.

The type of input accepted depends upon the module processing the data.

About the Core XIP Linguistic Units

In XIP, lexical nodes represent a single lexical reading of a linguistic unit. Each node is associated with features. A feature expresses a property of the node, such as part of speech, singular and plural, and so on. A feature consists of a name and a value pair.

For example, gender is a feature with the possible values of fem, masc, and neutral. Another feature is number, which has the possible values of plur and sing.

The features associated with nodes allow you to specify categories. A category is a collection of features. By convention, the category is named according to the part of speech value of one of its features. For example, a category that contains the noun, masculine, and plural features can be named the noun category.

About the Input Control Rules

As illustrated in Figure 1, between the pre-processing modules and the rule-based disambiguation module are the input control rules. The input control rules pre-treat the input text before it is processed by the disambiguation rules.

After the input control rules have been applied to the input text, the disambiguation rules can be executed. The disambiguation rules are described in the next section.

About Disambiguation

Part of speech disambiguation can be performed during the pre-processing phase using HMM taggers. XIP provides additional disambiguation facilities that can be used in the place of or in combination with the pre-processing disambiguator(s).

XIP uses a rules based disambiguation mechanism. Disambiguation rules choose the most likely reading(s) given the word's context. For example, the word help can be either a noun or a verb:

```
Many thanks for your help. //noun

Can I help you? //verb
```

Disambiguation rules can also be used to redefine feature values based on the context of the node.

About Chunking

The chunking module uses special chunking rules to group sequences of categories into structures that can be processed by the dependency module. The rules are organized in layers and are applied on sequences of categories one after the other on the disambiguated input text.

After processing is complete, XIP produces a chunk tree where contiguous and related words are united in chunks. For example, a verb phrase and a noun phrase chunk are identified as follows:

```
VP[he offers] NP[a nice present]
```

About Building Dependencies

The dependency module produces the dependency relations between words or chunks. These dependency relationships are specified by the grammar writer and may include:

- Standard syntactic dependencies, such as subject or object.
- Broader relationships, including inter-sentence relationships such as a co-reference.

Dependency relations are defined by a set of dependency rules that take as input a sequence of constituent or lexical nodes or a set of previously extracted dependencies. The constituent nodes can be constructed by the chunking module or by other external chunkers. Dependency rules are applied in sequence and rely on the evolving background knowledge stored in the chunk tree and in the dependency set.

For example, the following two dependency relationships can be found in the sentence "The dog eats soup":

```
Subject(eat, dog)

Object(eat, soup)
```

Introduction to a XIP Grammar

A XIP grammar consists of multiple text files that are used to disambiguate, chunk, and find dependency relations in a text. This section provides an overview of the grammar files and describes rules.

About the Grammar Files

A grammar contains the following basic types of files:

- Declarations of the tags used to describe features, categories, and dependencies in the XIP rules.
- Different types of rules written using operators and regular expressions to test the features of a node.
- A configuration file that specifies all of the files included in the grammar.

About Rules

Rules can specify a particular expression, a context, or constraints that are applied to the nodes. Rules are used to disambiguate the readings associated with a node, to construct the chunk tree, which groups nodes that share common properties and that work together as a unit, and to build dependencies between the nodes of the chunk tree.

The next chapter describes how data is represented in XIP and the basic structure of XIP rules.

Chapter 2:
Launching XIP Using the Command Line

To launch XIP, type the following at the command line:

```
nhxip –l english.xip -tr –f –text test.txt
```

The nhxip command launches the NTM and HMM pre-processing modules and XIP. The –l command specifies the grammar configuration file. The –tr command is a display command that restricts the number of features displayed. The –f command tells XIP to apply the dependency rules. The –text command specifies the file to be analyzed, in this example test.txt.

The following table describes the arguments you can use to execute XIP:

Argument	Description
-english	Specifies the English grammar configuration file. This file uses the strings.file so that the comments and commands will be in English.
-l *filename*	Provides the name of the grammar configuration file.
-text *filename*	Gives the text to analyze.
-number *value*	Overrides the Number field in the grammar configuration file.
-g	Specifies the grammar configuration file for a given language. This command differs from the –l command in that it specifies that the file is not in text file format.
-tagger	Specifies the tagging mode, when no parsing is applied.
-tagging	Specifies that disambiguation rules be used in the parsing.

Argument	Description
-ntagging	Specifies that disambiguation rules not be used in the parsing.
-f	Extracts the dependencies.
-x	Executes the XIP grammar without producing any output.
-max nb	Gives the maximum number of sentences that will be analyzed.
-trace	Generates a trace in the file defined in the grammar configuration file.
-indent	Generates the indented trees file.
-p *filename*	Provides the name of the file that contains the syntactic function that should not be displayed.

Displaying XIP Results

After launching XIP, you can display the output that results. This section describes the commands you can use to display information. For information about configuring the controls file, which contains information about how to display the XIP output and how features are handled, refer to the *XIP Reference Guide*.

The following table lists the commands you can use to display XIP output:

Argument	Description
-a	Displays all of the nodes.
-r	Displays a reduced set.
-tr	Displays even less node information.
-lem	Displays the same node set as the –tr command, but with the lemmas instead of the surface form.

Argument	Description
-t	Displays the chunks as a tree.
-ntree	Does not display the chunk tree. Instead it displays the dependencies with the sentence number as their first argument.
-xml	Displays the output as an XML entry.
-renum	Specifies that the word number start at 0 for each new sentence.
-nrenum	Specifies that the first word number of a new sentence continue from the numbers of the previous sentences.
-ixml	The input must be in XML. See –xml.

Chapter 3:
The XIP API – Reference Guide

You can use XIP as an external library that can be linked to any software that needs a linguistic analysis component. This chapter describes how to use the API and contains the following sections:

- Introduction to the API
- Initializing and Managing a XIP Grammar
- Parsing Information
- Returning Results
- System Management Objects
- API Basic Example

Introduction to the API

The XIP engine is written in C++, allowing you to easily integrate the parser as a library into any application that needs linguistic tools. The engine includes preprocessing components, such as a tokenizer, a morphological analyzer, and a built-in part-of-speech (POS) disambiguator.

Because the XIP parser sequentially processes a grammar, you do not need to use a complex algorithm to maintain concurrent analyses in memory. This reduces the memory footprint of the parser, because only one set of linguistic data is used throughout the entire analysis.

To avoid time lost when a rule is tried on a tree and fails, the XIP parser relies on the binary coding of features and syntactic categories.

Encoding Features

Each feature is coded as a bit position in a list of integers. The comparison of two sets of features is thus reduced to a comparison between a simple binary and two lists of integers that encode these features. Because the operations on the features are done on binaries, they are fast and highly portable.

Encoding Categories

Similarly, each category is coded as a bit on one integer. Thus, one integer encodes up to 64 different categories. This coding is useful for

filtering rules, because the filter pattern can be encoded in one integer. And, as for the features, binary operations are fast and highly portable.

Initializing and Managing a XIP Grammar

The following sections describe the functions available for loading a XIP grammar file, loading a parameter file, testing a grammar file, and freeing the grammar from memory.

Loading a Grammar

You can load a XIP grammar using the XipLoadGrammar() function as follows:

```
int XipLoadGrammar(char* grammar, int xml=0, char*
      ntmfile=NULL,char* hmmfile=NULL);
```

The parameters of the XipLoadGrammar() function are described in the following table:

Parameter Name	Description
char* grammar	This parameter corresponds to the path to the language file that will load the XIP grammar. For example, c:\english\english.xip.
int xml=0	When set to one, this parameter tells XIP that the input complies with the XIP XML DTD.
char* ntmfile	(Optional) This parameter corresponds to the path name of the NTM script. For example, c:\english\ntmscript.
char* hmmfile	(Optional) This parameter corresponds to the HMM file used to apply lexical disambiguation on the entries. For example, c:\english\english.hmm.

The XipLoadGrammar function returns a handler for a specific XIP grammar.

More than one grammar can be loaded in memory at a time. When you use more than one grammar, each grammar is referenced by a

unique handler returned by XIP. The remainder of this chapter refers to this handler as gHandler.

Loading a Parameter File

You can load parameter files in which refinements to the XIP core grammar have been made.

The XipParameterFile() function can be called multiple times when you have more than one parameter file that needs to be loaded. The XipParameterFile() function follows:

```
int XipParameterFile(int gHandler,char* filename);
```

The parameters of the XipLoadParameter() function are described in the following table:

Parameter Name	Description
int gHandler	The handler returned by the XipLoadGrammar function for the XIP grammar file being refined.
char* filename	The name of the parameter file you want to load.

Testing the Existence of a Grammar

You can test that a grammar exists using the XipExistGrammar() function as follows:

```
int XipExistGrammar(int gHandler);
```

In the function above, gHandler represents the grammar handler returned by the XipLoadGrammar function when the grammar was loaded.

If the grammar for the gHandler exists, the function returns 1. Otherwise, it returns 0.

Freeing a Grammar from Memory

You can free a grammar from memory using the XipFreeGrammar() function as follows:

```
void XipFreeGrammar(unsigned int gHandler);
```

The grammar referenced by the gHandler is freed from memory.

Parsing Information

The following sections describe the functions you can use to parse a text file and to parse a character string.

Parsing Files

XIP provides two functions for parsing text files: XipParseFileOS() and XipParseFile().

Using XipParseFileOS()

The XipParseFileOS() function stores the result of the parsed text file as an ostrstream object. The function is used as follows:

```
int XipParseFileOS(char* filename, int gHandler, ostrstream* os);
```

The parameters of the XipParseFileOS() function are described in the following table:

Parameter Name	Description
char* filename	The path and name of the text file being parsed.
int gHandler	The handler returned by the XipLoadGrammar function for the XIP grammar file.
ostrstream* os	The name of the ostrstream object that contains the result of the function.

Using XipParseFile()

The XipParseFile() function stores the result of the parsed text file as a XipResult object. The function is used as follows:

```
int XipParseFile(char* filename, int gHandler, XipResult* xip);
```

The parameters of the XipParseFile() function are described in the following table:

Parameter Name	Description
char* filename	The path and name of the text file being parsed.
int gHandler	The handler returned by the XipLoadGrammar function for the XIP grammar file.
XipResult* xip	The pointer to an existing XipResult object.

Parsing Strings

XIP provides two functions for parsing character strings: XipParseStringOS() and XipParseString().

Using XipParseStringOS()

The XipParseStringOS() function stores the result of the parsed character stream as an ostrstream object. The function is used as follows:

```
int XipParseStringOS(char* text, int gHandler, ostrstream* os);
```

The parameters of the XipParseStringOS() function are described in the following table:

Parameter Name	Description
char* text	A pointer to a character string.
int gHandler	The handler returned by the XipLoadGrammar function for the XIP grammar file.
ostrstream* os	The name of the ostrstream object that contains the result of the function.

Using XipParseString()

The XipParseString() function stores the result of the parsed character string as a XipResult object.

The XipParseString() function is used as follows:

```
int XipParseString(char* text, int gHandler, XipResult* xip);
```

The parameters of the XipParseFile() function are described in the following table:

Parameter Name	Description
char* text	A pointer to a character string.
int gHandler	The handler returned by the XipLoadGrammar function for the XIP grammar file.
XipResult* xip	The pointer to an existing XipResult object.

Returning Results

The result of a parse can be returned as a XipResult object. This object contains a vector of XipUnit objects. Each XipUnit object corresponds to the analysis of a given sentence. A XipUnit, in turn, is composed of the root of the chunking tree (a XipNode object) and a vector of dependencies (a XipDependency object).

Each XipNode object contains a set of features, stored as a vector of XipFeature objects. Each XipDependency is composed of a dependency name, a vector of XipNode objects, and a set of XipFeature objects.

Each object that represents XIP results provides its own print method to display the results on the screen.

The following sections describe how to use each object that can contain parsing results.

Storing Results Using the VECTA Template

The VECTA template is provided with the XipResult.h description file. The VECTA template is an abstract object that can transform pointers into vectors.

This template provides the different methods for accessing a given value. Each VECTA object has a last parameter that defines the number of elements stored in the vector. Each VECTA object also provides access to a given object through the redefinition of the square brackets ([]) operator.

Each element stored in the vector can be easily accessed. The following example illustrates how each dependency element of a XipUnit object can be accessed using the last parameter to define the limit of the vector:

```
XipUnit unit;



for (int i=0;i<unit.dependencies.last;i++)

   unit.dependencies[I]->print(cout);
```

This line of code iterates through the vector of dependencies until it reaches the last value. Then, it prints the output.

Using the XipResult Class

The XipResult class is the root of any analysis, whether the analysis is on a full text or a set of character strings. The XipResult object is composed of a vector of sentences and provides two different print methods. The XipResult class uses the following constructor:

```
class XipResult {
public:

  VECTA,XipUnit*>sentences;
  void print(ostream&,char feature=0);
  void printbare(ostream& os,char feature=0);

  ~XipResult();
};
```

Using the XipUnit Class

The XipUnit class stores the result of one linguistic unit, which can be a sentence or a complete paragraph. The XipUnit class uses the following constructor:

```
class XipUnit {
public:

  XipNode* root;
  VECTA<XipDependency*>dependencies;
```

```
  void print(ostream&,char feature=0);
  XipUnit();
  ~XipUnit();
};
```

A XipUnit is composed of a chunk tree, pointed to by the root field, and a vector XipDependency objects. A print method is provided to display the chunk tree together with the dependency graph.

Using the XipNode Class

The XipNode class stores the information related to a particular node in the chunk tree. The XipNode class also contains a vector of XipFeature objects that store the features assigned to a given node during the parsing process. The XipNode class uses the following constructor:

```
class XipNode {
public:

  char* surface;
  VECTA<char*> lemma;
  long left;
  long right;

  VECTA<XipFeature*>features;
  VECTA<XipNode*>daughters;

  void print(ostream&,char feature=0);
  void print_daughters(ostream&,char feature=0);

  XipNode(char* s,char* l, long g, long d);
  ~XipNode();
};
```

If the node is a non-lexical node (such as an NP or a VP), then the surface field stores the label of that node, while the lemma vector remains empty.

If a node is a lexical node, then the surface field stores the surface form of that lexeme, while the lemma vector stores all other possible reading for that lexeme.

For non-lexical nodes, the list of all the daughters of a node is accessible through the Daughters vector. Thus, you can explore the complete chunk tree using the root field of a XipUnit object as a starting point and scanning the daughter nodes.

For example, the following code displays all the daughter labels of the root node of the chunk tree:

```
XipUnit unit;


for(int i=0;i<unit.root.daughters.last;i++)
    cout<<unit.root.daughters[i]->surface;
```

Using the XipDependency Class

A dependency is a relation that links together nodes in the chunk tree. A dependency has a name, such as subject or object, and is associated with a set of features that are stored in a vector of XipFeature objects. The nodes that are linked together by a dependency are gathered in a vector of XipNode objects, each corresponding to a specific node in the chunk tree.

The XipDependency class uses the following constructor:

```
class XipDependency {
public:


  char* name;
  VECTA<XipFeature*>features;
  VECTA<XipNode*>parameters;
  void print(ostream&,char feature=0);


  XipDependency(char* n);
  ~XipDependency();
};
```

The XipDependency object does not store dependencies that have been hidden.

Using the XipFeature Object

The XipFeature class represents feature-value pairs. The XipFeature class uses the following constructor:

```
class XipFeature {
public:

   char* attribute;

   void print(ostream& os);

   XipFeature(char* a,char* v);
   ~XipFeature();
};
```

System Management Functions

The following functions can be used to manage your installation of XIP, including the library version and the license.

Managing XIP Libraries

The Whoami() function returns the version of the XIP library that you are using. The function is used as follows:

```
void Whoami(char* identity);
```

Verifying the XIP License

The XipLicense() function returns the number of days before the license associated with a given grammar expires. The function is used as follows:

```
int XipLicense(int ipar);
```

The XipLicense() function identifies the version of the library by returning the number of days before the license is expired.

API Basic Example

The following example illustrates how to use the XIP API to analyze sentences with the XIP grammar:

```
#include "xipresult.h"

//We allocate a XipResult object to store the result.
XipResult xip;

ghandler=XipLoadGrammar("c:\english\english.xip", 0,
                        "c:\english\ntmscript",
                        "c:\english\english.hmm");

//We parse the sentence
XipParseString("The dog drinks",ghandler,&xip);

//We display the result
xip.print(cout,0);

//We free the memory from the grammar.
XipFreeGrammar(ghandler);
```

Appendix A:
The XIP API - Include File

```
/*
 * Xerox Research Centre Europe
 *
 * Copyright (C) 2003 Xerox Corporation. All rights reserved.
 *
 * This file can only be used with the XIP library,
 * it should not and cannot be used otherwise.
 */
/* --- CONTENTS ---
   Project    : XIP
   Version    : 3.16
   filename   : xipstlres.h
   Date       : 10/01/2000
   Purpose    : Description of XipResult (STL version)
   Programmer : Claude ROUX
   Reviewer   :
*/

#ifndef result_h
#define result_h

//The different DISPLAY modes
#define DISPLAY_LEMMA 1
#define DISPLAY_SURFACE 2
#define DISPLAY_MARKUP 4
#define DISPLAY_ENTREE 8
#define DISPLAY_CATEGORY 16
#define DISPLAY_REDUCED 32
#define DISPLAY_FULL 64
#define DISPLAY_OFFSET 128
#define DISPLAY_WORDNUM 256
#define DISPLAY_SENTENCE 512
#define XML_NONE 0
#define XML_OUTPUT 1
#define XML_INSERT 2



#include <ostream>
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

#ifdef WIN32
```

```
#define Endl "\r\n"
#else
#define Endl "\n"
#endif


#ifdef XIPDLL_EXPORT
#define Exportation __declspec(dllexport)
#else
#define Exportation
#endif

#ifdef USEXIPLIBXML
#include <libxml/xmlmemory.h>
#endif


class XipFeature;
class XipNode;
class XipUnit;
class XipDependency;
class XipLeaf;

typedef vector<XipFeature* > VXipFeature;
typedef vector<XipNode* > VXipNode;
typedef vector<XipDependency* > VXipDependency;
typedef vector<XipUnit* > VXipUnit;
typedef vector<XipLeaf* > VXipLeaf;

typedef enum {XIPNODE,XIPLEAF} XIPRESTYPE;


class XipFeature {
public:

   string attribute;
   string value;

   Exportation void print();

   Exportation XipFeature() {};
   Exportation XipFeature(const XipFeature& x) {
      attribute=x.attribute;
      value=x.value;
   }


   Exportation XipFeature(string a,string v);
   Exportation ~XipFeature();
};


class XipNode {
public:

   string category;
```

```
   long left;
   long right;

   long index;

   VXipFeature features;
   VXipNode daughters;

   XipNode* mother;

   Exportation XipNode() {left=-1;right=-1;}
   Exportation XipNode(const XipNode& x);


   Exportation virtual void print(char feature=0);
   Exportation virtual void print_daughters(char feature=0);

   Exportation XipNode(string s,long g,long d);
   Exportation virtual ~XipNode();
   Exportation virtual XIPRESTYPE Type();
};


class XipLeaf : public XipNode {
public:

   string lemma;
   string surface;

   Exportation XipLeaf() {left=-1;right=-1;}
   Exportation XipLeaf(const XipLeaf& x);

   Exportation void print(char feature=0);
   Exportation void print_daughters(char feature=0);

   Exportation XipLeaf(string c, string s,string l, long g,long d);
   Exportation ~XipLeaf();

   Exportation XIPRESTYPE Type();
};


class XipDependency {
public:

   string name;
   VXipFeature features;
   VXipNode parameters;
   Exportation void print(char feature=0);

   Exportation XipDependency() {};
   Exportation XipDependency(const XipDependency& x);

   Exportation XipDependency(string n);
   Exportation ~XipDependency();
};
```

```
class XipUnit {
public:

   XipNode* root;
   VXipDependency dependencies;
   VXipNode leaves;

   Exportation void print(char feature=0);
   Exportation XipUnit(const XipUnit& x);

   Exportation XipUnit();
   Exportation ~XipUnit();
};

class XipResult {
public:

   Exportation XipResult() {};
   VXipUnit sentences;
   Exportation void print(char feature=0);
   Exportation void printbare(char feature=0);

   Exportation ~XipResult();
};


typedef void (*XipFunction)(int,XipResult*,void*) ;
//-------------------------------------------------------------------
//DLL exported functions
Exportation int XipParseStringOS(string text,int ipar, ostringstream* os,char
xml=XML_NONE,char apply_grammar=1);
Exportation int XipParseFileOS(string filename,int ipar, ostringstream* os,char
xml=XML_NONE,char apply_grammar=1);

Exportation int XipParseString(string text,int ipar, XipResult* xip,char
apply_grammar=1);
Exportation int XipParseFile(string filename,int ipar, XipResult* xip,char
apply_grammar=1);
Exportation int XipParseXipResult(XipResult* xipbase,XipResult* xipres,int ipar);
Exportation int XipParseXipLeaves(XipUnit* xipunit,XipResult* xipres,int ipar);
Exportation int XipParseXipNode(XipNode* xipnode,XipResult* xipres,int ipar);


Exportation int XipGrmFile(string grmname,int xml=0);

Exportation int XipLoadGrammar(string grammaire, int xml=0, string
ntmfile=NULL,string hmmfile=NULL);
Exportation int XipGrammarLoading(string grammaire, int xml=0, string
ntmfile=NULL,string hmmfile=NULL);
Exportation void XipFreeGrammar(unsigned int iParseur);
//This function reloads the grammar in memory.
Exportation int XipReloadGrammar(int ipar);

//Create a XipResult object
```

```
Exportation XipResult* XipResultCreate();
Exportation void XipResultDelete(XipResult* xip);

Exportation void Whoami(string& question);
Exportation int XipExistGrammar(int iParseur);
Exportation int XipParameterFile(int ipar,string filename);

//Return the number of days before the end of the license
Exportation int XipLicense(int ipar);
Exportation int XipIndentFile(int ipar,char open);

Exportation char* XipIndentFilePathname(int ipar);

#ifdef XIPLIBXML
Exportation int XipLoadXMLDataBase(int ipar, string filename,int depth);
Exportation int XipParseXMLFile(int ipar,string xmlfilename,int depth,XipResult* xip);
Exportation int XipParseXMLString(int ipar,string text,int depth,XipResult* xip);
//This function yields the current xmlNodePtr that is under treatment
#ifdef USEXIPLIBXML
Exportation xmlNodePtr XipXMLCurrentNode(int ipar);
#endif
Exportation char XipSetCallBackXML(int ipar,XipFunction f,void* data);
Exportation int XipParseStringXMLOS(string text,int ipar, ostringstream* os,int
depth,char xml=XML_NONE,char apply_grammar=1);
Exportation int XipParseFileXMLOS(string filename,int ipar, ostringstream* os,int
depth,char xml=XML_NONE,char apply_grammar=1);
#endif

//computeDependency==1, the dependencies are computed
//computeDependency=0, the dependencies are not computed
//the default is computeDependency=1, when XipLoadLibrary is called
Exportation void XipSetDependencyExtraction(int ipar,char computeDependency);

//Those two functions may be used to intercept a new result for one sentence
Exportation char XipSetCallBack(int ipar,XipFunction f,void* data);
//If the first XipResult is to be destroyed in the Call Back function, then
//the initial XipResult should be create with XipSetCurrentXipResult
//and deleted with XipDeleteCurrentXipResult
Exportation XipResult* XipCreateCurrentXipResult(int ipar);
Exportation char XipDeleteCurrentXipResult(int ipar);

//This function cleans the current XipResult object and
//reinitializes it.
Exportation XipResult* XipCleanCurrentXipResult(int ipar);

//This function retrieves the current XipResult object
Exportation XipResult* XipGetCurrentXipResult(int ipar);

Exportation void XipSetDisplayMode(int ipar, unsigned long mode);

#ifdef JMAX_DLL
//If jmaxonly=1 then only JMAX result is sent back
//By default jmaxonly=0
Exportation int XipParseFileJMAX(char* filename,int ipar, XipResult* xip,char
jmaxonly=0);
```

```
Exportation int XipParseStringJMAX(const char* text,int ipar, XipResult* xip,char
jmaxonly=0);
Exportation int XipLoadGrammarJMAX(const char* grammaire, const char* taf4,
const char* lsc, const char* tot);
Exportation int XipGrammarLoadingJMAX(const char* grammaire, const char* taf4,
const char* lsc, const char* tot);
#endif

#endif
```

Appendix B:
The XIP API - Example

```
/*
* Xerox Research Centre Europe
*
* Copyright (C) 2003 Xerox Corporation. All rights reserved.
*
* This file can only be used with the XIP library,
* it should not and cannot be used otherwise.
*/
/* --- CONTENTS ---
Project    : XIP
Version    : 4.09
Module     :
Date       :
Purpose    : Example of XIP library, using xipstl.dll
Programmer : Claude ROUX
Reviewer   :
Comment    : This example describes a connection to xipstl.dll and is based on STL
vectors.
*/

#include <stdlib.h>
#include <stdio.h>

#include "xipstlres.h"


void Usage()
{
   cerr<<"Usage:" <<endl;
   cerr<<"     -call"<<endl;
   cerr<<"     -grm grammar"<<endl;
}

//------------------------------------------------------------------------------

//-------------------------------------------------------------------------------

void DisplayNodes(XipNode* root,XipNode* node) {

/*
We first check the type of the node
XIPLEAF nodes store the different readings associated to a word.
A XipLeaf node provides the surface and the lemma of each reading.
while XIPNODE stores upper levels nodes such as phrasal nodes. A XipNode
only provides the category of the node.
   */
   if (node==root)
```

```
        cout<<"TREE"<<endl<<endl;
    if (node->Type()==XIPLEAF) {
        //It is a lexical node
        //We first recast the current node in a XipLeaf node
        XipLeaf* leaf=(XipLeaf*)node;
        cout<<leaf->surface<<":"<<leaf->lemma<<"<"<<leaf->left<<":"<<leaf-
>right<<">";
    }
    else {
        //It is not a lexical node
        cout<<node->category<<"<"<<node->left<<":"<<node->right<<">{";
        for (int i=0;i<node->daughters.size();i++) {
            if (i)
                cout<<",";
            DisplayNodes(root,node->daughters[i]);
        }
        cout<<"}";
    }

    if (node==root)
        cout<<endl;
}

//We display each dependency
void DisplayDependencies(VXipDependency& dependencies) {
    cout<<endl<<"DEPENDENCIES"<<endl<<endl;
//we loop on each dependency
    for (int i=0;i<dependencies.size();i++) {
        XipDependency* dep=dependencies[i];
        cout<<dep->name<<"(";
//for each parameter
        for (int p=0;p<dep->parameters.size();p++) {
            if (p)
                cout<<",";
            DisplayNodes(NULL,dep->parameters[p]);
        }
        cout<<")"<<endl;
    }
}

//We display each lexical node
void DisplayLeaves(VXipNode& leaves) {
    cout<<endl<<"LEAVES"<<endl<<endl;
    for (int i=0;i<leaves.size();i++) {
        DisplayNodes(NULL,leaves[i]);
        cout<<endl;
    }
}

//The call back function
void CallAfterEachSentence(int ipar,XipResult* res,void* data) {

    cout<<"-----------------------------------------------------------------"<<endl<<endl;
    cerr<<"We display a new sentence:"<<endl;
    //We display the result of the unique sentence, those functions are defined above
    DisplayNodes(res->sentences[0]->root,res->sentences[0]->root);
```

```
    DisplayDependencies(res->sentences[0]->dependencies);
    DisplayLeaves(res->sentences[0]->leaves);

    //After each sentence we clean the current XIPRESULT object to free the memory
    //If one has to parse a very large tree, then this method avoids the memory to
    //explode
    XipCleanCurrentXipResult(ipar);

}


//########################################################################
######################

int main (int argc, char *argv[]) {

    //This client needs a context, this line is alas mandatory...
    int i;


    string grmfilename;
    char callback=0;

    if (argc==1) {
        Usage();
        return -1;
    }

    for (i = 1; i < argc ;i++) {

        if (!strcmp(argv[i],"-call")) {
            callback=1;
            continue;
        }

        if (!strcmp(argv[i],"-grm")) {
            if ((i+1)>=argc) {
                cerr<<"Please choose a filename"<<endl;
                Usage();
                return -1;
            }
            else
                grmfilename=argv[++i];
            continue;
        }

        cerr<<"Unknown command:"<<argv[i]<<endl;
        Usage();
        exit(-1);
    }


    try {


        //We load a grammar together with a ntm script (finite-state lexicons).
```

```
        //The second parameter is set to 0 to tell the system that the initial sentence
        //(or file) is not an XIP XML result input. ipar is a handler on that grammar.
Since, more
        //than one grammar can be loaded at a time, we use this handler to distinguish
between
        //those different grammars.


        int ipar=XipGrmFile(grmfilename);

        //We decide to create a Callback function to intercept the XipResult objects built
for each sentence
        //being parsed.
        if (callback==1)
           XipSetCallBack(ipar,CallAfterEachSentence,NULL);

        //The input example.
        string phrs="They have decided to parse that sentence. The result is displayed
in two different ways.";

        //We create a XipResult handler
        XipResult* res=XipCreateCurrentXipResult(ipar);
        //We parse the string, the result is stored in "res"
        XipParseString(phrs,ipar,res,1);

        //In the case we don't have any call back function, we display all results at once.
        if (callback==0) {
           cout<<"We display all the results at once..."<<endl;
           cout<<"----------------------------------------------------------------"<<endl<<endl;
           for (i=0;i<res->sentences.size();i++) {
              DisplayNodes(res->sentences[i]->root,res->sentences[i]->root);
              DisplayDependencies(res->sentences[i]->dependencies);
              DisplayLeaves(res->sentences[i]->leaves);
              cout<<"----------------------------------------------------------------"<<endl<<endl;
           }
        }

        //We delete the last result
        char XipDeleteCurrentXipResult(ipar);

        //We delete the grammar
        XipFreeGrammar(ipar);
     }
     catch(char* message) {
        cerr<<message<<endl;
        exit(-1);
     }


     return 0;
}
```

Appendix C:
The XIP XML DTD

This appendix describes the XML DTD used by XIP. The XIP DTD is compliant with the W3C specification. This appendix contains the following sections:

- Defining XIP DTD Elements
- Defining XIP DTD Attributes
- Example XIP DTD

Defining XIP DTD Elements

The following tables describes the different elements available in the XIP DTD:

Element Name	Description
DEPENDENCY	Results from a linguistic analysis performed on the NODE elements. A DEPENDENCY contains two child elements: PARAMETER and FEATURES.
FEATURES	Provides the features of the DEPENDENCY. Note that these are not the attributes of the DEPENDENCY element itself because, in XML, an element cannot have a list whose length is not variable and that contains attributes whose names are not fixed.
LUNIT	Provides the linguistic unit, a list of nodes and a list of dependencies between these nodes. Note that because the definition of a node is recursive, the list of nodes is a list of chunk trees.
NODE	Contains the result of chunking (the morph-syntactic analysis performed by the chunking rules). A NODE structure is built upon the division in TOKENs.

Element Name	Description
	Because the definition of a NODE is recursive, NODEs can be trees. A NODE can be the parent of one TOKEN or several other NODES.
PARAMETER	Contains the words (or NODEs) between which a dependency has been established.
PCDATA	Provides a fragment of the input text, such as "small", "the", or "eat".
READING	Provides the disambiguated lexical unit.
TOKEN	Contains the result of tokenization, morphological analysis, and possibly lexical disambiguation. A TOKEN has three child elements: PCDATA, READINGS, and FEATURES.
XIPRESULT	Contains one of two child elements: a list of linguistic units (LUNIT) or a list of TOKENS.

Defining XIP DTD Attributes

This section describes the attributes contained by the various elements of the XIP DTD.

DEPENDENCY Attributes

The DEPENDENCY element has the attributes described in the following table:

Attribute Name	Description
name	Name of the dependency. This attribute is of type CDATA and is required.

FEATURE Attributes

The FEATURE element has the following attributes:

Attribute Name	Description
attribute	Name of the feature. This attribute is of

Attribute Name	Description
	type CDATA and is required.
value	Value of the feature. This attribute is of type CDATA and is required.

NODE Attributes

The NODE element has the following attributes:

Attribute Name	Description
num	A unique identifier for the node. This attribute is of type ID and is required.
tag	The name of the node. For example, this attribute could have a value of NP for noun phrase. This attribute is of type CDATA and is required.
start	Corresponds to the first word of the input text covered by the NODE structure. This attribute is of type CDATA and is required.
end	Corresponds to the last word of the input text covered by the NODE structure. This attribute is of type CDATA and is required.

PARAMETER Attributes

The PARAMETER element has the following attributes:

Attribute Name	Description
ind	An index. Each index corresponds to a NODE's number (num attribute). Therefore, the ind attriburte has a type of IDREF.
num	The number otf the PARAMETER in the DEPENDENCY. For example, the number is zero (0) if the parameter is the first argument in the dependency. This number is of type CDATA and is

Attribute Name	Description
	required.
word	When the NODE is a TOKEN, the word attribute is the PCDATA contained in this TOKEN. This attribute is of type CDATA and is required.

READING Attributes

The READING element has the following attributes:

Attribute Name	Description
lemma	The lemma being analyzed. This attribute is of type CDATA and is required.
pos	The part of speech. This attribute is of type CDATA and is required.

TOKEN Attributes

The TOKEN element has the following attributes:

Attribute Name	Description
start	Corresponds to the first word of the input text covered by the TOKEN. This attribute is of type CDATA and is required.
end	Corresponds to the last word of the input text covered by the TOKEN. This attribute is of type CDATA and is required.
surface	The surface form of the word described by the TOKEN. This attribute is of type CDATA and is required.
pos	The part of speech. This attribute is of type CDATA and is required.

Example XIP DTD

Following is an example XIP DTD:

```
<!ELEMENT XIPRESULT (LUNIT* | TOKEN*)>

<!ELEMENT LUNIT (NODE*, DEPENDENCY*)>

<!ELEMENT NODE (NODE*, TOKEN, FEATURES*)>
<!ATTLIST NODE        num     ID        #REQUIRED
          tag       CDATA #REQUIRED
          start     CDATA #REQUIRED
          end       CDATA #REQUIRED>

<!ELEMENT TOKEN (#PCDATA | READING | FEATURES )*>
<!ATTLIST TOKEN  start     CDATA       #REQUIRED
                 end       CDATA       #REQUIRED
                 surface CDATA       #REQUIRED
                 pos       CDATA       #REQUIRED>

<!ELEMENT READING (FEATURES?)>
<!ATTLIST READING          lemma       CDATA     #REQUIRED
                           pos         CDATA     #REQUIRED>

<!ELEMENT DEPENDENCY (FEATURES, PARAMETER*)>
<!ATTLIST DEPENDENCY       name        CDATA     #REQUIRED>

<!ELEMENT FEATURES (FEATURE*)>
<!ELEMENT FEATURE EMPTY>
<!ATTLIST FEATURE          attribute   CDATA     #REQUIRED
                           value       CDATA     #REQUIRED>

<!ELEMENT PARAMETER EMPTY>
<!ATTLIST PARAMETER        ind         IDREF     #REQUIRED
                           num         CDATA     #REQUIRED
                           word        CDATA     #REQUIRED>
```

Glossary of Terms

Category: a collection of features and their values. A category has a name that by convention refers to the part of speech value of one of its features. Each node in the chunk tree is associated with a category.

Chunk**: a linguistic unit identified by the chunking rules.**

Chunk Tree: a tree structure that groups nodes together that share common properties and work together as a unit.

Chunking Rules: rules that group sequences of categories into structures (chunk tree) that can be processed by the dependency module. There are two types of chunking rules: ID/LP rules and sequence rules.

Controls File: contains information about how to display the XIP output and how features are handled.

Dependency: a linguistic relation between one or more words.

Dependency Rules: rules that calculate dependency relationships between nodes. They have the following basic format:
|*pattern*| if <*condition*> <*dependency_term*>.

Disambiguation: a linguistic service that finds the correct grammatical category of a word according to its context.

Disambiguation Rules: rules that disambiguate the category of a word. They have the following format:
layer > *readings_filter* = |*left_context*| *selected_readings* |*right_context*|.

Features: characterize categories. For example, gender is a feature with the possible values of fem, masc, and neutral.

Finite-State Transducer: see *FST*.

FST: Finite-State Transducer. An FST is a network of states and transitions that work as an abstract machine to perform dedicated linguistic tasks.

Grammar Configuration File: a text file that defines the other files that compose the grammar used by XIP. For example, english.xip.

Hidden Markov Model: see *HMM*.

HMM: Hidden Markov Model. An algorithm used for part of speech disambiguation.

ID Rules: rules that describe unordered sets of nodes. ID rules have the following format:
layer> new_node -> list_of_lexical_nodes.

ID/LP Rules: Immediate Dominance/Linear Precedence Rules. These rules identify sets of valid categories. See also *ID Rules* and *LP Rules*.

Immediate Dominance Rules: see *ID Rules*.

Linear Precedence Rules: see *LP Rules*.

LP Rules: work with ID rules to establish some order between the categories. They have the following format:
layer> [*sequence of categories*] < [*sequence of categories*].

Marking Rules: rules that mark specific node configurations in the chunk tree.

Mother Node: a node that is built on top of a sequence of nodes. For example, NP might be the mother node to a determiner and a noun.

Node: the basic building block of the chunk tree. Lexical nodes are associated with lexical units. Non-lexical nodes are associated with a group of lexical nodes.

Parameter file: file used to refine the core grammar. This file contains custom rules that are appended to the core grammar.

Percolate: when the features of a daughter node are shared with the mother node.

Reshuffling Rules: rules that rebuild sections of the chunk tree. The left of a reshuffling rule describes an initial sequence of sub-trees (as described in marking rules) and a right side that specifies modifications to the chunk tree.

Sequence Rules: rules that describe an ordered sequence of nodes. They have the following format:
layer> new_node = list_of_lexical_nodes.

Sibling Node: nodes that are related but do not share the same mother.

Sister Node: nodes that share the same mother node.

Terminal Feature: feature specified using braces ({}).

Terminal Set: describes a set of one or more lexical readings associated with a given token.

Tokenization: the isolation of word-like units from a text.

xfst: Xerox Finite-State Toolkit, which is used to create finite-state networks, including the lexical transducers for morphological analysis and generation.