Xerox Incremental Parser

Reference Guide

6, CH DE MAUPERTUIS
38240 MEYLAN
FRANCE

xerox

Printed in France

Contents

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 provides reference materials for describing XIP data, formulating XIP rules, and implementing a XIP grammar.

Audience

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

Users of the API appendix should have experience programming in C++.

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:

- Describing XIP Data:
  Provides information about describing the data processed by XIP. This chapter includes information on declaring features, categories, and dependencies, as well as how to define the other types of XIP grammar file.
- Formulating XIP Rules:
  Provides general information about formulating rules, including variables and operators, and specific information about disambiguation, chunking, and dependency rules.

- Implementing a XIP Grammar:
  Describes how to run XIP from the command line and how to refine the XIP grammar using parameter files.
- Appendix A, Using the XIP API:
  Describes the classes available in the XIP API, which you can use with any software that needs a linguistic analysis component.

Other XIP Documentation

In addition to this manual, the documentation set for XIP includes the *XIP User's Guide*, which describes in detail how to write your own XIP grammar. It includes information about the XIP architecture and an introduction to the parts of a XIP grammar.

Chapter 1:
Describing XIP Data

This chapter explains how to describe the data used by XIP. It contains the following sections:

Common Attributes of the Grammar Files

This section describes common attributes of the XIP grammar files. It includes a list of all the possible types of grammar files.

List of Grammar File Types

A XIP grammar consists of a set of configuration files and a set of rule files. The following tables describes the files of a XIP grammar:

File Type	Description
Feature declaration file	Contains the declarations for all of the features associated with categories. See "Declaring Features" on page 12 for more information.
Category declaration file	Contains the declarations for all of the categories used by XIP. See "Declaring Categories" on page 11 for more information.
Dependency declaration file	Contains all of the declarations for the dependency tags used by the

File Type	Description
	dependency rules. See "Making Free and Bound Features" on page 36 for more information.
Custom lexicon file	Contains lexical rules that append new features to existing words, modify or replace categories, or add new categories for existing words. See "Defining Custom Lexicons" on page 15 for more information.
Controls file	Contains information about the automatic features of XIP, including features that are appended to nodes and substrings used to split input text into a sequence of categories. See "Defining Feature Defaults and Display" on page 19 for more information.
Translation file	Contains translation rules for translating external tags to XIP tags when they come from an external source. See "Translating External Tags" on page 23 for more information.
Grammar configuration file	Defines all of the files that compose your grammar. See "Creating a Grammar Configuration File" on page 24 for more information.
Strings file	Contains the English language field names used in the grammar configuration file. See "Creating a Grammar Configuration File" on page 24 for more information.

Several files are created during the execution of a XIP grammar. They are described in the following table:

File Type	Description
Trace file	Provides the complete trace for the execution of the XIP application.
Indentation file	Gives the name of the trace file that represents the chunk tree using indentations. One space is added for each node level. This file is useful for displaying nodes along with their features.

Overview of the General Syntactic Rules

This section describes the characters used to write comments and escape characters in a grammar file.

Comment Characters

Comments can be added anywhere in XIP. You can write a comment that spans one or more lines using the slash (/) and backslash (\) characters as follows:

```
/this is a comment\
```

You can also add comments to the end of a line using two slash characters (//) as follows:

```
NP = det,noun. //this is a comment
```

Escape Characters

XIP uses the backslash (\) character to escape characters that might be misinterpreted by the internal parser, such as a blank space or an equal sign (=). For example, the following lexicon declaration for the string “in front” contains a blank character that has been escaped:

```
in\ front = prep.
```

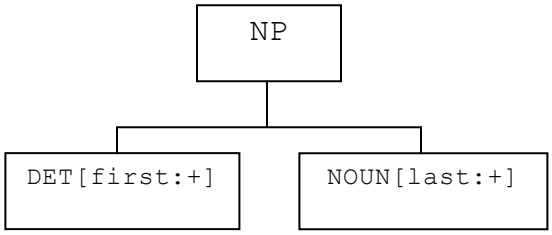
About Automatic Features

XIP automatically appends some features on specific nodes. The system allows you to define the names of these features with the exception of the first, last, start, and end features, which cannot be modified. These automatic features are described in the following sections.

First and Last Features

The first and last features are automatically instantiated on the first and last node of a chunk, which is defined in the grammar. They can be

tested in rules to verify the correct position of a node. For example, a noun phrase node consists of the following nodes:



The DET and NOUN nodes automatically receive the first and last features. If the chunk were reduced to a single node, it would receive both the first and last features.

Start and End Features

The start and end features are automatically instantiated on the first and the last node of a sentence or any input text.

For example, start and end features are associated with nodes in a text as follows:

```
The[start:+] lady opens the big door[end:+]
```

If the last token of the sentence is a period (.), then the end features is attached to it as follows:

```
The[start:+] lady opens the big door.[end:+]
```

You can percolate start and end features. For more information, refer to "Using V-Rules to Modify Features" on page 39.

About Automatic Filtering

XIP automatically filters certain features to avoid displaying unnecessary features to the user. These filters are formatted in the controls file using the Display, FunctionDisplay, and NodeDisplay fields. Refer to "Defining Feature Defaults and Display Options" on page 19 for more information about configuring these fields.

Declaring Categories

Categories correspond to nodes in the chunk tree. A category is a name associated with a collection of features.

For information on declaring categories associated with free or bound features, refer to "Making Free and Bound Features" on page 36.

Syntax of Category Declarations

Categories are declared in the category declaration file using the following format:

```
Categories:
    noun=[cat=noun].
    verb=[cat=verb].
```

The name (usually a part of speech) is given on the left of the equal sign (=) and the set of features for the name is given on the right of the equal sign.

Declaring the Root Node

The virtual root node is the first category defined in the category declaration file. For example, the following category declaration defines a virtual root node of top:

```
Categories:
    top = [top=+]
    noun = [nominal=+, verbal=-, bar=1]
    verb = [verbal=+, nominal=-, bar=1]
```

Declaring Features

Features express properties of nodes. A feature is expressed as a feature:value pair.

Syntax of Feature Declarations

Features are declared in the feature declaration file using the following format:

```
Features:
  Root:[
      a1:{v1,v2},
      a2:[
         a3:{v3,v4},
         a4:{v5,v6}
        ]
      ]
```

Rules for Defining Features

The following rules apply to the definition of a collection of features:

- The feature tree has one root that is always followed by a square bracket ([), such as Root:[.
- A colon (:) always follows a feature name.
- A feature is defined over a list of features using the [*list_of_features*] notation.
- A feature is defined over a list of values using the {*list_of_values*} notation.
- Feature definitions are separated by a comma (,).
- A feature cannot be defined over a mix of features and values.
- If the definition of a feature reduces to one value, the braces ({}) are optional.

The following example illustrates the use of these rules in the formulation of a feature definition:

```
Features:
[root:
  [agreement:
    [gender:{fem,masc,neutral},
     number:{plur,sing},
     pers:{1,2,3}
    ],
   subcat:{subNP,subPP},
   nominal:+,
   verbal:{+}
  ]
]
```

For information on free and bound features, refer to "Making Free and Bound Features" on page 36.

Declaring Dependencies

XIP finds dependency relationship between nodes in the chunk tree. Each dependency is identified with a tag that must be declared.

Syntax of Dependency Declarations

Dependencies are declared in the dependency declaration file using the following format:

```
Functions:
  subject.
  object.
  NN.
```

Dependency tags must be declared in this file. You can declare as many dependency tags as you want.

Features of dependencies must be declared in the feature declaration file. To display the dependencies with their features, they must also be declared in the controls file. For more information about the features file, refer to "Declaring Features" on page 12. For more information about the controls file, refer to "Defining Feature Defaults and Display Options" on page 19.

Hiding Dependencies

Some dependencies should be computed but not displayed in the result. These dependencies do not need to be deleted, only hidden.

To declare a dependency tag as hidden in the dependency declaration file, use the following format:

```
Hidden:
   subj, obj.
```

Dependencies that appear in the Hidden section must also appear in the Functions section.

Dependencies that are hidden are not displayed and are not stored in the XipDependency object. For more information about this object, refer to "Using the XipDependency Class" on page 72.

For example, a dependency declaration file contains the following:

```
Functions:
  subject.
  object.
Hidden:
  object.
```

The sentence “The lady opens the door” yields the following result:

```
subject(opens,lady)
```

The object dependency is not displayed because it has been declared in the Hidden field.

Defining Custom Lexicons

You can define custom lexicons that add new features that are not stored in the standard lexicon. The lexicon file also lets you define translations for markup language tags (such as HTML and XML) to improve the precision of parsing.

The first line in a lexicon file must contain the following:

```
Vocabulary:
```

This line tells the XIP engine that the file contains a custom lexicon.

Lexical Rule Format

Lexical rules attempt to provide a more precise interpretation of the tokens associated with a node.

Syntax

Lexical rules have the following syntax:

```
lemma(: POS([features])) (+)(+)= (POS)[features].
```

The parts of the rule contained by parentheses (()) are optional. The optional operators are described in “Operators in Lexical Rules” on page 16.

The POS is optional but must be used when you want to test features.

Some example lexical rules follow:

```
dog:noun  += [animate=+]
Mr        =  noun[human=+,title=+]
Xerox     += verb[transitive=+]
in\silico =  adv
```

In the above example, new features are added to the nouns dog and Mr. Xerox is given the additional reading of verb, and in\silico is added as a new adverb.

How Lexical Rules Work

Lexical rules can either test features or test the part of speech (POS). XIP processes lexical rules as follows:

1. All entries are removed and replaced by a single solution.
2. Specific entries are modified on the basis of their features or their POS.
3. A new interpretation is appended to the list of interpretations.

Operators in Lexical Rules

The following table describes operators you can use in formulating lexical rules:

Operator	Description
:	Compares the features of nodes to provide a set of comparisons. This operator applies only to features whose domain of declaration is composed of features and not of terminal values. For example: #1[agreement]:#2[agreement] This comparison succeeds if the agreement feature of node #1 contains some of the same values as the agreement feature of node #2.
::	Imposes a strict comparison between features. This operator requires all fields to be instantiated in both structures. For example:

Operator	Description
	#1[agreement]::#2[agreement] This comparison succeeds if node #2 bears the same values for the agreement feature as node #1.
< <= > >=	Specifies a domain over which the value for a given feature should be valid. For example: #1[num] < [3] This comparison means that the value 3 should be part of the domain of the num feature. #1[num] < #2[num] This comparison means that the value of num for #1 is less that the the value of num for #2.
~:	Verifies that two nodes bear different values for the same feature. This operator can be used with all of the previous operators. For example: #1[agreement]~:#2[agreement] This test succeeds if nodes #1 and #2 have different values for their agreement features.

Modifying the Lemma or Surface Form

The lemma and surface form can be modified using the += operator in two ways. You can identify a lemma followed by optional part of speech information as follows:

```
dog:noun[+] += = verb[+].
```

You can also indicate simply the lemma and the new lemma form as follows:

```
name += noun[lemma=thename].
```

The name noun now has a new lemma form, thename.

In both, the information to the left of the += operator is used to test the lemma and the information on the right is the new values that will modify the lemma.

Defining Markup Language Tags

XIP provides a mechanism for taking into account markup language tags (such as HTML and XML). The tags you want to be processed must be declared in a lexicon file.

Two sorts of tags are distinguished, opening tags and closing tags. The opening tag triggers a mechanism that instantiates a specific set of features to all of the following words until the closing tag is found. The operators for declaring an opening or closing tag are described in "Operators in Lexical Rules" on page 16.

The markup tag features can be tested in other rules as normal features. For example, in HTML documents, the <H1> tag is used to mark a title string. The presence of a title feature that is induced by this tag could drive the selection of nominal analysis over verbal analysis when a word is ambiguous. For example, the lexicon file contains the following lexical rules:

```
Vocabularies:
<H1> <= [title=+].
</H1> >= [title=+].
```

When the <H1> tag is first encountered, the [title=+] feature is added to the general feature assigned to every lemma. When the </H1> tag is encountered, the [title=+] feature is removed from the general features list.

You can also use a toggle operator when a tag has only one form. For example, the lexical file contains the following lexical rule:

```
Vocabularies:
" <=> <firstword:+>[quote=+].
```

The first time the quote (") is found, the firstword feature is added to the general feature list. The second time the quote is found, it is

considered as a closing tag and the firstword feature is removed from the general list.

Markup language tags can includes an asterisk (*) sign as a wildcard character. For example, the following tag applies even if the <BODY> tag is followed by additional information, such as background color, text color, and cell padding:

```
<BODY *>
```

Defining Feature Defaults and Display Options

XIP provides default features and feature display options that are used during the parsing of a text. The default features and display options are described in the controls file. The following sections describe each field of the controls file.

Tag Field

The Tag field is used in conjunction with the translation rules described in “Translating External ” on page 23. When the parsing of a list of syntactic tags fails to find a part of speech, then the value of this field is used as a default.

The value of this field is not a XIP part of speech but a string that will be translated into a XIP part of speech.

For example, the Tag field of the controls file contains the following:

```
Tag:
  NOUN
```

The NOUN string will be used as the default string that will be translated into a XIP part of speech when no part of speech has been identified for an entry.

Boundaries Field

The Boundaries field defines the list of strings used as linguistic input boundaries. The input of XIP is a list of entries that the parser splits into sequences of words and then applies the grammar to these sequences. The value of this field is the string that is systematically searched for in an entry to verify if the entry should be used as an end of sentence or paragraph.

The Boundaries field does not contain any notion of part of speech or feature. XIP simply looks for this string in the entry.

For example, the Boundaries field of the controls file contains the following:

```
Boundaries:
  +SENT
```

The +SENT string will be looked for in every entry output by the morphological analyzer. For example:

```
The    the+DET
lady   lady+Noun
opens  open+P3+VERB
the    the+DET
door   door+NOUN
.      .+SENT  //The +SENT string is detected in this entry.
The    the+DET
dog    dog+NOUN
```

The parser splits the above list of words into two linguistic units.

Lemma Field

The Lemma field defines the feature name used to reference the lemma value of a node

For example, the Lemma field of the controls file contains the following:

```
Lemma:
  [lemma]
```

The value of the Lemma field does not need to be different from the field name.

Surface Field

The Surface field defines the feature name used to reference the surface value of a node.

For example, the Surface field of the controls file contains the following:

```
Surface:
  [surface]
```

The value of the Surface field does not need to be different from the field name.

For example, for the word “dogs”, the lemma value is dog and the surface value is dogs.

Display Field

The Display field lists the features displayed in the indented file or in the trace file. This field contains a list of features.

The indented file displays the results of the chunking module as an indented tree. For each node of the tree, the list of features is built and displayed. The Display field allows you to focus the display of these values.

For example, the Display field of the controls file contains the following:

```
Display:
  [gender, plural]
```

Only the gender and plural features will be displayed in the XIP output.

FunctionDisplay Field

The FunctionDisplay field lists the features that will be displayed with the dependency name.

XIP creates a chunk tree and a list of dependencies. A list of features can be attached to a XIP dependency. For example, the difference between a subject and a passive subject in XIP consists of using the same dependency name, in this case subject, with a specific passive feature attached to the passive subject.

Features listed in this section must be declared in the feature declaration file described in “Declaring Features” on page 12.

For example, the FunctionDisplay field of the controls file contains the following:

```
FunctionDisplay:
  [passive, reflexive, possessive]
```

When the sentence “The door is opened by the lady” is analyzed, the following result is displayed:

```
subject_passive(open,lady)
```

The passive feature is displayed after the dependency name with an underscore (_).

NodeDisplay Field

The NodeDisplay field describes the list of features displayed after a node in the chunk tree and in the dependencies.

Features listed in this section must be declared in the feature declaration file described in "Declaring Features" on page 12.

For example, the NodeDisplay field of the controls file contains the following:

```
NodeDisplay:
  [gender,number]
```

When the sentence "The lady talks" is analyzed, the following output is displayed:

```
Subject(talks_SING, lady_FEM_SING)
S{NP{the lady_FEM_SING} VP{talks_SING}}
```

The feature values declared in the NodeDisplay field are displayed with an underscore (_) before them.

Uppercase Field

The Uppercase field declares the automatic features that are set when a word starts with an uppercase character.

Features listed in this section must be declared in the feature declaration file described in "Declaring Features" on page 12.

For example, the Uppercase field of the controls file contains the following:

```
Uppercase:
  [upper:+]
```

When the sentence "The lady enters the Palace" is analyzed, both "The" and "Palace" receive an upper:+ feature.

AllUppercase Field

The AllUppercase field declares the automatic features that are set when a word is written in all uppercase characters.

Features listed in this section must be declared in the feature declaration file described in "Declaring Features" on page 12.

For example, the AllUppercase field of the controls file contains the following:

```
AllUppercase:
  [allupper:+]
```

When the sentence “The lady enters the PALACE” is analyzed, only the word “PALACE” receives the allupper:+ feature.

Translating External Tags

You can use the output of external tokenization and morphological analyzers as input for the XIP disambiguation module. For example, you might use the NTM module to normalize, tokenize, and morphologically analyze text. For more information about the NTM module, refer to the *XIP User’s Guide.*

However, the input provided by external sources may contain tags that vary from those used by XIP. XIP provides a translation file for translating external tags to XIP tags so that the grammar does not need to change when different external input is used.

Syntax of Translation Rules

The translation file contains the translation rules. It uses the following format:

```
Translation:
  NounProper=noun[proper=+].
  Sg=[sg=+].
  Pl = [pl=+].
```

On the left of the equal sign is the part of speech or feature from the external lexicon. On the right is the translation into XIP format. Every translation rule must end with a period (.).

Every tag from the external lexicon should have a translation in this file. If a tag is not translated in this file, the parser simply skips the unknown tag. No error message is issued unless the –warning command has been selected on the command line. For more information about using the command line, refer to “Using XIP Commands” on page 58.

Example Translation Rules

The following translation rule translate one noun POS tag to another:

```
Translation:
  Nom = noun.
```

The following translation rule translates a feature tag to a feature value declared in the list of features:

```
Translation:
  FEM = [gender=fem].
```

The following translation rule translates a tag to a part of speech with a specific set of features:

```
Translation:
  DAY = noun[time=+,day=+].
```

Creating a Grammar Configuration File

The grammar configuration file ties together all the other files of the XIP grammar. A grammar consists of the text files described in this section and the rules used to disambiguate, chunk, and build dependency relations in a text.

The grammar configuration file stores the basic parameters of the system and describes the files that compose the grammar of a given language. By convention, the grammar configuration file is given the name of the language of the grammar, such as english.xip for the English grammar.

Description of the Fields in the Grammar Configuration File

The following sections describe each field of the grammar configuration file.

License Field

The License field contains the name of the license holder.

Modules Field

The Modules field describes the XIP modules for which the license is valid. By default, the modules field lists the following modules:

Module Name	Description
chunker	Creates the chunk tree.
inference	Implements the dependency calculus.

Module Name	Description
extraction	Extracts dependences. This legacy dependency calculus module is present for compatibility reasons.
semantic	Disambiguates the text.
server	Transforms XIP into a server.

Expiration Field

The Expiration field provides the license expiration date in year/month/day format. This format makes the expiration date easier to compare with the current date.

Code Field

The Code field contains a key computed from the License, Modules, and Expiration fields. This checking key confirms that the system was installed correctly..

Language Field

The Language field gives the default language of your system. The language you specify here determines the language of the field names in the grammar files. The field names are provided by the strings.file for English and the chaines.fic for French. If the string file is not present in your XIP directory, it will be recreated automatically.

By default, the Language field is set to French. To add a new language, you need to modify one of the string files.

Locale Field

The Locale field was used in previous versions of XIP to specify the character set of the language described by this grammar. This field is present for legacy purposes and is no longer used by the latest versions of XIP.

Number Field

The Number field tells the system the maximum number of terminal features (features specified using {}) present at compilation time. The number of features is always computed over a multiple of 64, which is the size of an integer in bits (one integer can store up to 64 different values). The number of integers that will store all the different values must be specified at the beginning of compilation.

For example, a value of 5 imposes a maximum of 64 x 5 = 320 different values declared in the grammar. The default value is 5. If this value is too small, the compiler will return an error message asking you to set the Number field to a higher value.

Indentation Field

The Indentation field gives the name of the file that contains an indented tree representation of the chunk tree. An indented tree displays each node along with its list of features.

Trace Field

The Trace field gives the name of the file that contains the complete trace for the execution of the XIP application. Each rule that has been applied to parse a sentence is recorded in sequence, so that the complete processing of a linguistic unit can be traced.

Features Field

The Features field contains a list of the files used to describe the complete feature system of the grammar. The features, categories, tag translations, dependencies, and features controls can all be contained in one file, or split across many files as described in the previous sections.

Lexicons Field

The Lexicons field provides the name of the custom lexicon file(s). For information about creating a custom lexicon file, refer to "Defining Custom Lexicons" on page 15.

Rules Field

The Rules field lists the rules files applied to the linguistic input. XIP applies the rules according to their order in this field.

Each rule file contains a set of rules, each recorded in a specific layer. The layer numbers can be absolute or relative to the previous file that was compiled. A layer number is relative to the last layer compiled if the file name is preceded by a plus (+) in this field.

For example, the rules1.xip file contains the following rule:

```
1> NP = Det,?*, Noun.
```

The rules2.xip file contains the following rule:

```
1> VP = Pron, Verb.
```

The layer numbers of these rules are absolute if they are listed in the Rules field of the grammar configuration file as follows:

```
Rules: rules1.xip, rules2.xip
```

When the rule numbers are absolute, they are processed in the same layer as follows:

```
1> NP = Det,?*, Noun.
1> VP = Pron, Verb.
```

The layer numbers of these rules are relative if they are listed in the Rules field of the grammar configuration file as follows:

```
Rules: rules1.xip, +rules2.xip
```

When the rule numbers are relative, they are processed in the two separate layers as follows:

```
1> NP = Det,?*, Noun.
2> VP = Pron, Verb.
```

Example Grammar Configuration File

Following is an example grammar configuration file:

```
/XIP information and license details\
License: Xerox Research Centre Europe
Modules: chunker, inference, extraction, semantic, server

/Expiration date\
Expiration: 2010/11/31

/check key for verifying the license\
Code: xyz012345abc

/Localization fields\
Language: english
Locale:en

Number: 7

/Output files\
Indentation:trees.out /Indented tree output\
Trace:trace.out   /Traces for the XIP application\

/The following files define the basic linguistic components used in the grammar\
Features:features.xip,categories.xip,translations.xip,controls.xip, functions.xip

/Custom lexicons\
Lexicons: lexicon.xip


/Grammar rules\
/File containing LP-rules, Sequence rules and ID rules\
/The + symbol signifies that the layer numbers are relative\
Rules: adjust.xip, +localgram.xip, +chunker.xip, +dependency.xip
```

Chapter 2:
Formulating XIP Rules

This chapter provides a reference for formulating the different types of rules. It includes the following sections:

Using Variables

All rules can associate nodes with variables to test specific values or compare nodes on the basis of certain features.

Syntax of Variables

A variable is declared in a rule when the definition of the node is followed by:

```
#number
```

The *number* must be greater than or equal to one. Variables are local to a rule. They allow you to compare nodes on the basis of certain features.

You assign a variable by placing it directly after the name of the node. For example, consider the following rule:

```
NP = Det,Noun#1
```

#1 is the variable associated with the Noun node.

Operators Used with Variables

The following sections describe the operators that can be used with variables to test a feature or compare nodes.

[] Operator

The [] operator tests the set of features on a node. This operator can be used with the other operators described in this section.

For example, the following test succeeds when node #1 bears the feminine and plural features:

```
#1[gender:fem,number:pl]
```

:: Operator

The :: operator verifies that two variables represent the same node. For example, the following rule tests that the #1 node and the #2 node are the same:

```
#1::#2
```

This operator imposes a strict comparison between features, when combined with the square brackets ([]) operator. For example, the following test requires all features to be instantiated in both structures:

```
#1:[agreement]::#2[agreement]
```

If the agreement feature of node #1 bears the gender and number features, then the agreement feature of node #2 must also bear these features for the test to succeed.

: Operator

The : operator compares the features of nodes. Unlike the :: operator, which imposes a strict comparison, this operator compares the common attributes and values and does not require all fields to be instantiated in both structures.

For example, the following test succeeds if node #1 and node #2 bear some common values for the agreement feature:

```
#1:[agreement]:#2[agreement]
```

~: Operator

The ~: operator verifies that two variables correspond to different nodes. For example, the following test succeeds when #1 and #2 are different nodes:

```
#1~:#2[agreement]
```

< Operator

The < operator verifies that one node precedes another in the chunk tree. For example, nodes in a sentence are associated with variables as follows:

```
The lady#1 opens the door#2.
```

The following test succeeds, because lady precedes door in the chunk tree:

```
#1 < #2
```

You can also use the < operator to compare if the features on the left are less than the features on the right. For example, the following test compares the numerical features of variables #1 and #2:

```
#1[num] < #2[num]
```

You can also identify a value that you want the feature to be less than. For example, the following test succeeds if the num feature of variable #1 has a value less than four:

```
#1[num] < 4
```

> Operator

The > operator verifies that one node follows another in the chunk tree. For example, nodes in a sentence are associated with variables as follows:

```
The lady#1 opens the door#2.
```

The following test fails, because lady precedes door in the chunk tree:

```
#1 > #2
```

You can also use the > operator to compare if the features on the left are more than the features on the right. For example, the following test compares the numerical features of variables #1 and #2:

```
#1[num] > #2[num]
```

You can also identify a value that you want the feature to be greater than. For example, the following test succeeds if the num feature of variable #1 has a value more than three:

```
#1[num] > 3
```

~ Operator

The ~ operator can be used with all of the previous operators. It indicates difference.

For example, this operator can be used to verify that two nodes bear different values for the same feature. The following test applies a loose comparison of features using the : operator:

```
#1[agreement]~:#2[agreement]
```

The test succeeds if the two nodes have different values for some of their agreement features.

The following test applies a strict comparison of features using the :: operator:

```
#1[agreement]~::#2[agreement]
```

The test suceeds if the two nodes have different values for all of their agreement features.

= Operator

Variables can also be used to share features between distant nodes. The = operator makes the transfer of features possible. For example, the following rule states that the gender of node #1 is the intersection of the elements of nodes #2 and #3:

```
#1[gender]={#2 & #3}.
```

Boolean Expressions Used with Variables

Variables can be used in Boolean expressions that compare the features of specific nodes. Node tests can be combined with the Boolean operators "or" (|) and "and" (&). Areas for interpretation can be constrained using parentheses (()).

The following test uses a Boolean expression:

```
(#1[gender:fem] | #1[gender:masc])
```

The expression above tests if the gender is feminine *or* masculine.

Boolean expressions are evaluated from left to right. For example, the following tests are equivalent:

```
(#1[gender:fem] & #1[gender:masc] | #1[number:plur])

(#1[gender:fem] & (#1[gender:masc] | #1[number:plur]))
```

Notice how parentheses can be used anywhere to constrain the interpretation of the Boolean expression.

Different tests can be mixed with the Boolean expressions, as illustrated in the following example:

```
(#1[number:sing] & #1[gender]::#2[gender])
```

The above expression tests that node #1 bears the value sing, and that nodes #1 and #2 share the same value for the gender feature.

Using Tree Regular Expressions

A tree regular expression (TRE) is a special type of regular expression that establishes connections between distant nodes. TRE expressions explore the inner structure of sub-nodes. Braces ({}) are used in a rule to indicate that sub-nodes must be examined. An example TRE follows:

```
NP{det,noun},FV{verb}.
```

Using Variables in TREs

Assuming that the head of a chunk is its last element, we can isolate this head using variables instead of using its category name.

For example, the following TRE applies to any NP and FV spanning any kind of categories:

```
NP{?*,#1[last]},FV{?*,#2[last]}.
```

This rule records the matching nodes for the two variables, #1 and #2, which correspond to the heads of each of their mother nodes. Because this rule applies to any kind of NP, we can constrain the rule to extract only nouns as follows:

```
NP{?*,Noun#1[last]},FV{?*,#2[last]}.
```

Variables, when used without any categories, are equivalent to the "any" symbol (?).

TRE Operators

TREs use several special operators that are described in the following sections.

; Operator

The semi-colon (;) operator is used in TRE to indicate "or". For example, the following rule uses the semi-colon operator:

```
NP{det,noun[last]}; NP{pron}, FV{verb}.
```

The above rule looks for a sub-tree that starts with a NP that spans a det and a noun or a NP that spans a pron followed by a FV.

* Operator

The asterisk (*) operator is used in TRE to indicate "one or more". For example, adj* specifies zero or more adj categories.

The asterisk operator and the semi-colon operator can be used in combination, just as for other regular expressions.

? Operator

The question mark (?) operator is used in TRE to indicate "any". For example, the following rule uses the question mark operator:

```
NP{?*,#1[last]}, ?*, FV{?*, #2[last]}.
```

The above rule looks for a sub-tree that starts with a NP that is followed by a FV on its right, anywhere.

Exploring Sub-Nodes for a Category

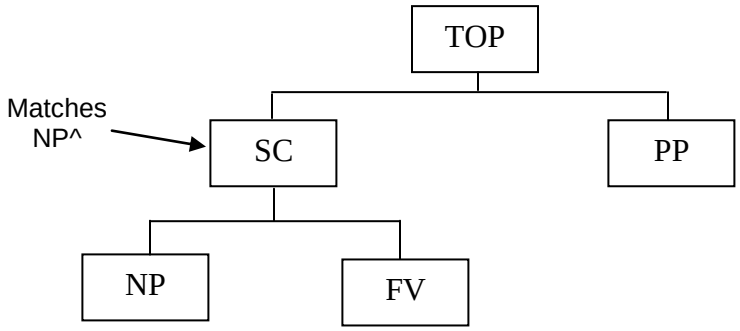
You may want to constrain a node on a category that can be present at the same level or below the node in the chunk tree. Sub-nodes are searched using the following expression:

```
Cat^
```

The category matches the current node if the current node is a *Cat* or if the current node contains a *Cat* at any depth under it. For example, a rule is written as follows:

```
NP^, PP.
```

This rule looks for a node that is a NP or contains an NP in its sub-tree followed by a FV. This rule matches the following chunk tree:



Computations on Features

During rule analysis, actions are taken on the feature:value pairs associated with features.

Computations Summary

Following is a list of possible actions that rules can take on features:

Feature Computation	Action
[feature:value]	This value should be present on the node or unification fails.

Feature Computation	Action
[feature=value]	The feature is set with the value. Unification fails if the feature is already set with a different value.
[feature:?]	The feature should have a value on the node or the unification fails.
[feature]	The feature should have a value on the node or unification fails.
[feature:~]	The feature should not be instantiated on the node or unification fails.
[feature:~*value*]	The feature should not have the *value*.
[feature=~]	The feature is cleared of all values on the node.
[!feature=value]	The feature is set with the value and freed.
[!feature:value]	The feature must have the value and is then freed.
[!feature:!]	The feature is freed, no matter what value it bears.
[feature:!]	The feature is bound.

Specifying Local or Global Features

There are two ways to manage the interpretation of words belonging to several categories:

- Different features stemming from all of the interpretations can be merged into one, global features set.
- Each specific set of “local” features can be addressed individually.

Global Features

The global set of features results from merging all of the different features that result from the different interpretations of a word. Brackets ([]) are used to test the global feature set.

For example, the word help has the following interpretations:

```
help    +verb

help    +noun
```

The global feature set of the word help is [noun:+,verb:+]. The different features are merged into a unique set of features.

You can use brackets to test features that have been instantiated on the global set of features. For example, the following expression tests a noun that has a verbal interpretation:

```
noun[verb:+]
```

Local Features

Each word that has more than one interpretation is associated with as many sets of features as it has interpretations. Each set of features associated with an interpretation of a word is called a local set of features. You can test each set of local features separately using angle brackets (<>).

For example, the word help has the following interpretations:

```
help    +verb

help    +sing+noun
```

The word help has the following local features:

```
[verb:+] //first set of local features

[noun:+,sing:+] //second set of local features
```

You can test a local features set as follows:

```
noun<sing>
```

The test checks if the noun interpretation of this word contains the singular feature.

Making Free and Bound Features

Features can be free or bound. The values of free features are shared between the daughter node and its mother node. This sharing of features from lower nodes to upper nodes is called percolation. When

a free feature percolates to a mother node, it is unified with the features on this node. If the unification fails, the application of the rule also fails.

The values of bound features do not percolate from a daughter node to its mother node.

Declaring Free and Bound Features

Feature status can be specified during category declaration as described in the following table:

Declaration	Description
[!feature=value]	The feature is set with the value and freed.
[!feature:value]	The feature must have the value specified and is then free.
[!feature:!]	The feature is freed, no matter what value it contains.

Making Features Free or Bound in Rules

You can free or bind features in rules. For example, compare the following rules:

```
np -> det, noun[gender=fem],adj. //rule 1

np -> det, noun[!gender=fem],adj. //rule 2
```

In rule 1, only a noun with the gender:fem feature can be accepted. If the node does not bear any gender feature, then the rule will apply and the fem value will be instantiated on the node.

In rule 2, the exclamation mark (!) means that this value should become free. A free value is a value that is freely instantiated from mother node to mother node.

You can stop the percolation of the np node in two ways:

- np[gender:!]
- np[gender:masc]

When the node is unified with one of these categories, the gender feature will stop acting as a free feature and the feature will be bound.

Example of a Free Feature

For example, the following rule is applied to a sentence:

```
np = det,noun[!number:!].
```

In this example, we suppose the number feature is a free feature on the noun node.

We use the following rule to parse the following sentence:

```
The ladies open the door.
```

The first noun phrase receives a number feature from the noun node automatically as follows:

```
S{NP[number:pl]{the ladies[!number:pl]},VP{open},
        NP[number:sing]{the door[!number:pl]}}
```

Negating Features

The tilde (~) operator negates features. Features can be negated in two ways:

- feature:~value
- feature:~

For example, the following expression indicates that a node should not bear any values for the gender feature:

```
noun[gender:~]
```

The following expression indicates that a node cannot bear a feminine value for the gender feature:

```
noun[gender:~fem]
```

Feature negation can be used in V-rules to test the absence of a value for a given feature. For more information about V-rules refer to "Using V-Rules to Modify Features" on page 39.

Lemma Testing

You can test the value of a lemma in a sequence of categories. To test the lemma value, use the lemma feature defined in the Lemma field

of the controls file. For more information about this file, refer to "Defining Feature Defaults and Display Options" on page 19.

The lemma feature is used to test the lemma value of a node as follows:

```
verb[lemma: eat]
```

In the above example, only the verb node that is associated with the word eat will be accepted.

Using V-Rules to Modify Features

Valence rules (V-rules) provide a way of adding features based on other features. There are two types of V-rules: default features specification (DFS) rules and feature co-occurrence restriction (FCR) rules.

All V-rules can be either pre or post. Pre V-rules are applied before the instantiation of free features from the daughter nodes on the mother nodes. Post V-rules are applied after the instantiation of free features from the daughter nodes to the mother node.

DFS Rule Syntax

A DFS rule is composed of two lists of features. The list on the left provides a condition and the list on the right provides the set of features that should be instantiated. The failure to instantiate features does not cause the analysis to fail.

An example of a DFS rule follows:

```
[pron:+, indefinite:+] > [nominative=+].
```

This rule instantiates the nominative value on the node if the node contains the pron and indefinite features.

FCR Rule Syntax

FCR rules have the same syntax as DFS rules. However, they are applied differently from the DFS rules. If the left part of the FCR rule applies, then the right part of the FCR rule must also apply. Therefore, the application of an FCR rule can cause the analysis to fail.

Percolating Start and End Features with PreDFS

XIP automatically instantiates the start and end features on the first and last node of the sentence.

These features are used locally to verify the position of a node in the sequence. You can transform these features into free features that

percolate until the end of chunking by adding the following PreDFS rule:

```
PreDFS:

  [start:+]>[!start=+]

  [end:+]>[!end=+]
```

Any upper node built on a node carrying percolating start and end features will also carry these two features.

Split Rules

Split rules break the input stream into processing units. Split rules are processed sequentially from right to left.

The keyword for identifying split rules in a rules file is SplitRules.

The following split rule breaks input after a colon if a verb is found on the left side of a colon:

```
SplitRules:

  |VERB, ?*[punct:~], punct[form:fcolon]|.
```

The following split rule breaks the input whenever a SENT tag occurs:

```
|SENT|.
```

Split rules are applied after the lexical rules as part of the input control rules applied before rule-based disambiguation. For more information about lexical rules, refer to "Defining Custom Lexicons" on page 15. For more information about how input control rules are processed, refer to the *XIP User's Guide*.

Disambiguation Rules

Disambiguation rules disambiguate the most likely category(s) given a word's context.

Syntax of Disambiguation Rules

A disambiguation rule consists of the following tests:

```
layer> readings_filter = |left_context| selected_readings |right_context|.
```

The following sections describe each part of the rule in detail.

Layer Number

Layers are used to mix disambiguation rules with other sorts of rules, such as ID/LP rules.

Each layer is written as a number between one and 300 followed by the greater-than sign (>).

A rule with no layer number is automatically inserted in layer 0, a layer that is applied before any other rules of any kind are applied.

Readings Filter

The left side of a disambiguation rule contains the readings filter. The readings filter specifies a subset of categories and features that can be associated with a word. The list can be constrained with features. As described in "Specifying Local or Global Features" on page 35, use square brackets ([]) to test global features and angle brackets (<>) to test local features.

If the filter contains more than one element and tests a global set of features, then parentheses (()) should be used.

The filter applies when it matches a subset of the complete ambiguity class of a word. The filter can use the question mark (?) operator to refer to any part of speech.

Following are some example readings filters:

```
//rule 1:
noun,verb                          //The readings contain a noun or verb.

//rule 2:
(noun,verb)[adj:~]          //The reading contains a noun or a verb.
                                   The global set of features cannot contain
                                   an adjective feature, meaning this word
                                   cannot have an adjective reading.

//rule 3:
noun,verb,~adj              //Equivalent meaning to rule 2. The
                                   reading contains a noun or a verb.
                                   The global set of features cannot contain
                                   an adjective feature, meaning this word
                                   cannot have an adjective reading.

//rule 4:
noun<sing>,verb             //One of the local readings of the noun
                    should be singular.

//rule 5:
(noun<sing>,verb)[adj:~]    //One of the local readings is a singular
```

```
                                    noun.  The word can also be a verb.  None
                                    of the readings is an adjective.

//rule 6:
noun[adj:~]                         //Among all the readings of this word, one
                                    is a noun.  None of the readings is an
                                    adjective.  This is equivalent to
                                    (noun)[adj:~].
```

The reading filters are applied to the word light. An example of the application of reading filters follows:

```
light

                    light +Adj
                    light +Sing+Noun
                    light +Verb

                    Local set of features:
                                        noun:[sing:+,noun:+]
                                        verb:[verb:+]
                                        adj:[adj:+]

                    Global set of features:
                                        [sing:+,noun:+,verb:+,adj:+]
```

Rules 1 and 4 apply, because the filter matches a subset of the readings of light. Rules 2, 3, 5 and 6 do not apply because the global set of features contains the value [adj:+].

Left and Right Contexts

The left and right contexts contain a sequence of nodes. A context is always written between pipes (||). A context can be negated with a tilde (~) symbol before the first pipe.

Selected Readings

The selected readings portion of a disambiguation rule gives the selected interpretation(s) of the word.

Using Operators and Keywords in Disambiguation Rules

This section describes the special operators and keywords used in disambiguation rules.

<> Operator

You can use angle brackets (<>) in disambiguation rules to define specific features associated with a category. For example, the following context refers to a singular noun:

```
|noun<sing:+>|
```

[] Operator

You can use brackets ([]) to refer to the complete set of features for a category. For example, the following context indicates any category that is not a verb:

```
|?[verb:~]|
```

% Operator

The percent sign (%) operator restricts the interpretation of a word to one solution. This operator can also be used to replace the readings for a word with a new reading.

For example, the following rule transforms a verb beginning with an uppercase letter to a noun, even if the word does not have a noun reading:

```
verb[uppercase]%= noun[uppercase=+].
```

The following rule states that if a word precedes a non-ambiguous noun that has an adjective reading, then remove all other readings but the adjective reading. The word is then disambiguated as an adjective.

```
noun[verb:~,adj:~] = |det,adj%| noun.
```

The rule above is applied on the following sentence:

```
The light balloon goes up.
```

The word light will be disambiguated as an adjective, because balloon can be a noun only.

<* Operator

You can use the asterisk (*) character with angle brackets to specify that each reading must bear the features listed after the asterisk. For example, the following context indicates that all noun readings must bear the case:acc features:

```
noun<*case:acc>
```

where Keyword and Variables

Variables can be used in disambiguation rules to further constrain the application of a disambiguation rule. For example, the following disambiguation rule contains variables and the where keyword:

```
verb = |pron#2[nominative]|verb, where(#0[pers]::#2[pers]).
```

A word that has a verb reading will be disambiguated as a verb if there is a pronoun on its left that shares the same person feature.

The rule is applied to the following sentence:

```
She lights the fire.
```

The word light is disambiguated as a verb.

Example Disambiguation Rules

The following disambiguation rule states that the determiner reading is preferred for a word that has any category except for a noun or a pronoun and is followed by any number of adjectives and a noun. This rule does not apply to quantifiers.

```
det[quant:~] = ?[noun:~,pron:~] |adj*,noun|.
```

The following rule states that the determiner reading of a word is selected when it is followed by any number of adjectives and a noun that does not contain the adverb or verb features:

```
det<quant>,pron = det |adj*,noun[adv:~,verb:~].
```

The following rule identifies coordinated numerals:

```
num = num |coord*,num%|.
```

The following rule removes the numeral reading when a word also contains a determiner reading:

```
num,det = ?[num:~].
```

The following rule saves only the noun and adjective readings for a word when the ambiguity class of a word is noun or verb and the immediate left context contains a determiner:

```
noun,verb = |det| noun,adj.
```

A readings filter does not need to match the complete set of possibilities for a word, but only a sub-set of the possibilities. However, the selected reading must match the categories that will be kept exactly.

The above rule could be rewritten as follows:

```
noun,verb = |det| ?[verb:~].
```

This rule saves any category that does not bear the verb feature.

Chunking Rules

Chunking rules group sequences of categories into structures that can be processed by the dependency module. There are two types of chunking rules: immediate dependency and linear precedence rules (ID/LP rules) and sequences rules. This section describes how chunking rules use layers and the use of the two types of chunking rules.

Layers in Chunking Rules

Each chunking rule must be defined in a specific layer, numbered between 1 and 300. This layer number is written as a number followed by a less than (>) sign as follows:

```
1> NP = det, noun.  //layer 1

2> VP = pro, verb.  //layer 2
```

Layers are processed according to their rank, from the first layer (1) to the last. Each layer can contain only one type of chunking rule.

ID/LP Rule Structure

ID rules describe unordered sets of nodes. LP rules work with ID rules to establish some order between the categories.

Syntax of ID Rules

ID rules have the following syntax:

```
layer> node_name -> list_of_lexical_nodes.
```

An example of an ID rule follows:

```
NP -> det,noun,adj.
```

This rule defines creates an NP node for any set of categories that contains a determiner, a noun, and an adjective, in any order. This rule applies to the following phrases:

```
the blue bird

the bird blue
```

Syntax of LP Rules

LP rules are associated with ID-rules. They can apply to a particular layer or be treated as a general constraint throughout the XIP grammar. LP rules have the following syntax:

```
layer> [set_of_features] < [set_of_features].
```

The layer number is optional.

For example, the following LP rule states that every adjective precedes a noun in all ID rule:

```
[adj:+] < [noun:+]
```

The following LP rule states that a determiner must precede a noun only in layer 2:

```
2> [det:+] < [noun:+]
```

Operators in ID/LP Rules

ID/LP rules can use parentheses (()) to accept optional categories. They can also use the Kleene star convention (*) to indicate that zero or more instances of a category are accepted.

NOTE: You cannot use the question mark (?) symbol in the right side of ID/LP rules to indicate any category.

For example, the following rule states that the determiner is optional and that as many adjectives as possible are accepted:

```
NP -> (det),noun,adj*.
```

This rule applies to the following phrases:

```
bird

blue bird

the blue bird

the big blue bird
```

Contexts in ID/LP Rules

ID/LP rules can be constrained with contexts. For example, the following rule defines two contexts:

```
NP -> |det,?*| noun,adj |?*,verb|.
```

The first context specifies that a determiner must be on the left of the selected set of categories. The second context specifies that a verb must be on the right side of the set of categories. We apply the rule on the following sentence:

```
The blue bird eats the crumbs.
```

We obtain the following chunk:

```
NP[blue,bird].
```

Example ID/LP Rules

Our grammar contains the following ID rules:

```
2> NP-> det, adj, noun.

2> NP -> det, noun.

2> NP -> noun.
```

These rules are applied to the following sentence:

```
The blue bird sings in the tree.

/This sentence consists of: det adj noun verb prep det noun\
```

XIP extracts the following valid sequences of categories:

- The first sequence isolated is “the tree”, because XIP begins by analyzing the sentence from right to left.

Rule two applies, because the sequence is composed of two categories, det and noun.

- The second sequence extracted is “The blue bird”. Because the verb and the preposition do not occur in any of the rules, no sequence containing these parts of speech will be built out of the sentence.

In this case only one rule applied, the one that contains the largest set of parts of speech:

```
NP -> det, adj, noun.
```

Sequence Rule Structure

Unlike ID/LP rules, which work on unordered groups of nodes, sequence rules describe an ordered sequence of nodes. A sequence rule is deterministic and, by default, extracts the shortest list of categories that matches the rule.

Syntax of Sequence Rules

Sequence rules have the following syntax:

```
layer> node_name = list_of_lexical_nodes
```

Contexts in Sequence Rules

Sequence rules can contain a context that constrains where a node can occur. For example, the following sequence rule defines two contexts:

```
NP = |det,?*| noun |?*,verb|.
```

This rule is applied to the following sentence:

```
The blue bird eats the crumbs.
```

The result of the sequence rule follows:

```
NP[bird].
```

Operators in Sequence Rules

You can impose a longest match reading to a context using the at sign (@) operator. Extracting the longest list of categories that match a rule can be useful when a disambiguation takes place in the context itself. For example, the following sequence rule reaches the most remote determiner on the left:

```
NP = |@ det,?*| noun.
```

You can apply sequence rule to the longest phrase using the @= or @<= operators. For example, the following sequence rule applies from left to right and finds the longest match:

```
1> NP @= det,?*[verb:~],noun.
```

You can negate a whole set of categories and features or a category within a set of categories using the tilde (~) operator. For example, the following sequence rule negates a set of categories:

```
NP = ~|det, adj| noun.
```

The above rule indicates that a determiner followed by an adjective must not occur before the noun. To negate just the determiner category, the sequence rule can be written as follows:

```
NP = |~det, adj| noun.
```

In this sequence rule, the noun must not be preceeded by a determiner but must be preceeded by an adjective.

The question mark (?) operator can be used to represent any category.

The asterisk (*) operator, also known as the Kleene star convention, can be used to indicate zero or more categories.

Optional categories can be indicated using parentheses (()). For example, the following rule indicates that the adj category is optional:

```
NP = det, (adj), noun.
```

Disjunction can be indicated using the semi-colon (;) operator. For example, the following rule indicates that det can be followed by one of the categories listed (adj, coord, or adv):

```
NP = det, adj;adv;coord, noun.
```

Variables and the where Keyword in Sequence Rules

Sequence rules can use variables and the where keyword.

If the where keyword follows a category, it should be preceded by a comma (,). If it follows a context, then no character is needed. For example, the following sequence rules use variables and the where keyword:

```
NP = det,?*,noun#1, where(#1[number:plur]). //needs a comma

NP = det,?*,noun#1 |?*,verb| where(#1[number:plur]). //nothing needed
```

Contexts can contain variables to be compared with other elements of the rule. For example, the following contexts use variables:

```
NP = det,?* noun#1 |?*,verb#2| where(#1[pers]::#2[pers]).
```

Examples of Sequence Rules

The example sequence rules described in this section are applied to the following sentence:

```
The blue bird eats the crumbs.
```

Following is an example of a basic sequence rule:

```
NP = det, ?*, noun.
```

When applied to the sample sentence, this rule yields the following:

```
NP[the blue bird]

NP[the crumbs]
```

The following sequence rule imposes constraints on the features:

```
NP = det, ?*, noun[number:pl].
```

When applied to the sample sentence, this rule yields the following:

```
NP[the crumbs]
```

The following rule includes variable tests:

```
SC = noun#1,?*,verb#2, where(#1[pers]::#2[pers]).
```

When applied to the sample sentence, this rule yields the following:

```
SC[bird eats]
```

The following sequence rule looks for the longest sequence of matching categories instead of the shortest:

```
NP @= det,?*,noun.
```

When applied to the sample sentence, this rule yields the following:

```
NP[the blue bird eats the crumbs].
```

The following sequence rule uses a context:

```
NP = det,?*,noun |?*,verb|.
```

When applied to the sample sentence, this rule yields the following:

```
NP[the blue bird]
```

Dependency Rules

Dependency rules take the sequences of constituent nodes identified by the chunking rules and identify relationships between the nodes.

Syntax of Dependency Rules

Dependency rules have the following syntax:

```
|pattern| if <condition> <dependency_terms>.
```

The *pattern* contains a TRE that describes the structural properties of parts of the input tree. The *condition* is any Boolean expression built from dependency terms, linear order statements, and operators.

The *pattern* and *condition* are both optional. For more information about TREs, refer to " Using Tree Regular Expressions" on page 32. For

more information about Boolean expressions, refer to "Boolean Expressions Used with Variables" on page 32.

Operators in Dependency Rules

The following sections describe the special uses of operators in dependency rules.

^ and ~ Operators

When the carat (^) operator is introduced before a dependency name, the result of the rule will be the modification or deletion of the marked dependency. When the carat (^) operator is used to mark a dependency, the rule applies in a deterministic way to avoid more than one modification applying at a time on a given dependency.

The dependency is deleted when the result of a rule is the tilde (~) operator.

For example, the following sentence is analyzed:

```
The dog eats the soup in the house.
```

The following set of dependencies are extracted:

```
1) Subj(eat,dog)
2) Comp_Obj(eat,soup)
3) Comp_Loc(eat,house)
```

The following dependency rule transforms the Subj dependency into a SComp dependency:

```
if (^Subj(#1,#2) & comp(#1,#3))  SComp(#1,#2,#3).
```

This rule applies only on dependencies 1) and 2) because the ^ operator introduces a deterministic behavior on the selection of the next dependency. The dependency rule results in the following:

```
SComp(eat,dog,soup).
```

Deterministic Operators (|| and &&)

You can also impose determinism in dependency rules by doubling the ampersand (&) and pipe (|) operators. The double operator improves the efficiency of rule processing. When you need to verify the existence of at most one dependency, this operator reduces the size of the analysis to the first dependency to solve the rule. The rest of the rule is not explored.

For example, the following dependency rules are identical except for the use of the pipe operator:

```
if(subj(#1,#2)&(Comp[Obj](#1,#3)|Comp[Loc](#1,#3))SComp(#1,#2,#3).

if(subj(#1,#2)&(Comp[Obj](#1,#3)||Comp[Loc](#1,#3))SComp(#1,#2,#3).
```

We apply the first rule to the following set of dependencies:

```
1) Subj(eat,dog)
2) Comp_Obj(eat,soup)
3) Comp_Loc(eat,house)
```

The rule applies in a non-deterministic way and yields the following results:

```
SComp(eat,dog,soup)
SComp(eat,dog,house)
```

If we apply the second rule on the same set of dependencies, the || operator behaves as a deterministic operator. It will stop at the first dependency that complies with the rule, yielding the following dependency:

```
SComp(eat,dog,soup).
```

Zone Operators

Dependency rules can be defined in specific zones. These zones can be activated or deactivated to apply to a specific set of rules. Zones that are not active do not apply during parsing time.

Zones must be defined in the parameters file. For more information about the parameters file, refer to "Refining the Grammar with Custom Definitions" on page 61.

The zones operator can be introduced anywhere in a dependency rule. For example, the following zone operators divide a series of rules into zones:

```
<zones:*>
//The rules below belong to all zones.
subject(#2,#1)=NP[fonc:~,fonc=subj]{?*,#1[last]},FV{?*,#2[last]}.
subject(#2,#1)=NP[fond:~,fond=subj]{?*,#1[last]},?*,FV{?*,#2[last]}.
```

```
<zones:1,2,3>
//The rules below belong to zones 1, 2, and 3.
object(#2,#1)=FV{?*,#2[last]},NP[fonc:~,fonc=subj]{?*,#1[last]}.
object(#2,#1)=FV{?*,#2[last]},?*,NP[fonc:~fonc=subj]{?*,#1[last]}.

<zones:1>
//The rule below belongs to zone 1.
if (subject(#2[!pers:!],#1) & object(#2,#3)) SVO(#1,#2,#3).
```

Example Dependency Rules

This section provides a variety of examples of dependency rules.

Creating a New Dependency

The following dependency rule extracts a subject-verb-object dependency for any noun phrase that has already been bound to a subject dependency and an object dependency:

```
|NP{?,#1[last]}|if(subject(#2,#1)&object(#2,#3))SVO(#1,#2,#3).
```

The following dependency rule creates a subject-verb-object dependency when the condition is verified:

```
if(subject(#2,#1)&object(#2,#3)) SVO(#1,#2,#3).
```

You can create more than one dependency with a single dependency rule. For example, the following dependency rule creates both a subject and an object dependency:

```
|SC{NP{?*,#1[last]}, VP{?*,#2[last]}}, NP{?*,#3[last]}|
            Subj(#2,#1), Obj(#2,#3).
```

Modifying a Dependency

The following dependency rule marks the subject dependency for modification:

```
|NP{?,#1[last]}|if(^subject(#2,#1)&object(#2,#3))SVO(#1,#2,#3).
```

If this rule applies, the subject dependency is replaced by the SVO dependency.

Percolating Features from Nodes

The following dependency rule instantiates the pers value on the new dependency, SVO:

```
if(subject(#2[!pers:!],#1)&object(#2,#3))SVO(#1,#2,#3).
```

Deleting a Dependency

The following dependency rule deletes the object dependency if all of the conditions are true:

```
|NP{?,#1[last]}|if(subject(#2,#1)&^object(#2,#3))~.
```

The object dependency is marked with the carat (^) operator and the result of the rule is the tilde (~) operator.

The following dependency rule deletes the object dependency when the condition is verified:

```
if(subject(#2[!pers:!],#1)&^object(#2,#3))~.
```

Adding Features to a Dependency

When the subject dependency contains a passive feature, the following dependency rule copies this feature to the SVO dependency:

```
|NP{?,#1[last]}|if(subject[!passive:!](#2,#1)&comp(#2,#3))SVO(#1,#2,#3).
```

For example, the following dependency set is extracted for the sentence “The soup is eaten by the bird”:

```
subject_PASSIVE(eaten,soup)
comp(eaten,bird)
```

After applying the rule, the subject_PASSIVE dependency contains the following features:

```
subject_PASSIVE(soup,eaten,bird)
```

Testing Features on a Dependency

The following rule results in an SVO dependency if the condition on the passive feature of the subject dependency is verified:

```
if(subject[passive](#2,#1)&comp[by](#2,#3)) SVO(#1,#2,#3).
```

Testing Features on a Node

The following rule tests the features of the #2 node:

```
if(subject(#2[passive],#1)*comp[by](#2,#3)) SVO(#1,#2,#3).
```

If the node bears the passive feature, the result is an SVO dependency. Free features can percolate from the tested nodes to the new dependency.

Rules for Modifying the Chunk Tree

The rules described in this section can be used to modify the chunk tree created by the chunking rules.

Marking Rules

A marking rule is a simple TRE that instantiate specific features on nodes that match a sub-tree. For more information about TRE, refer to " Using Tree Regular Expressions" on page 32.

For example, the following marking rule instantiates the passive feature on the FV node when the FV node spans the be auxiliary followed by a past participle verb:

```
FV[passive=+]{aux[be],verb[ppe]}.
```

For example, this rule is applied to the following sentence:

```
NP{the door} FV{is opened}
```

The FV receives the passive feature.

Reshuffling Rules

Reshuffling rules rebuild sections of the chunk tree. They can be used in layers and may be mixed with chunking rules. Like marking rules, reshuffling rules use TRE.

Reshuffling rules are composed of a TRE on the left side of the rule and another TRE that describes the new sub-tree on the right. For more information about TRE, refer to " Using Tree Regular Expressions" on page 32.

An example reshuffling rule follows:

```
PP#1{?*#2, NP#3{?*#4,#5[last]}} = #1{#2,#4#5}.
```

The rule flattens a prepositional phrase that contains a noun phrase and a list of elements under the noun phrase. For example, we apply the rule to the following phrase:

```
PP{with NP{the lady}}
```

The two nodes are rebuilt by the reshuffling rule as follows:

```
PP{with the lady}
```

If the right part of a reshuffling rule provides a new part of speech for a variable, then the new part of speech replaces the previous part of speech. For example, the following reshuffling rule suggests a new part of speech:

```
PP#1{?*#2,NP#3{?*#4,#5[last]}} = NP#1{#2,#4#5}.
```

We apply this rule to the following phrase:

```
PP{with NP{the lady}}
```

The two nodes are rebuilt by the reshuffling rule as follows:

```
NP{with the lady}
```

Chapter 3:
Implementing a XIP Grammar

This chapter provides reference details about the files and commands required to implement a XIP grammar. It contains the following sections:

- Using XIP Commands
- Refining the Grammar with Custom Definitions

Using XIP Command

The following sections describe the commands used to launch XIP, display data, and monitor the execution of the grammar.

Execution Commands

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.
-I *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.
-tagger	Indicates tagging mode, where no parsing is applied.
-tagging	Indicates that the disambiguation rules are used in the parsing.

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

Display Commands

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 of nodes.
-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.
-t	Displays the chunk tree.
-tc	Individually displays the sub-nodes under the top node

Argument	Description
-tl number	Displays the chunk tree, and wraps it at "number" characters.
-ntree	Only displays the dependencies (no tree)
-obn	Orders by Node. The dependencies are ordered along their first argument.
-xml	Displays the output as an XML entry.
-balise	Displays the XML tags.
-renum	Sets the word number to 0 for each new sentence.
-nrenum	Indicates the first word number of a new sentence continues from the numbers of the previous sentences.
-sent	Displays only the sentences with their number as they are processed.
-x	Executes without producing output.

Trace Commands

The following table lists the commands you can use to monitor the execution of XIP:

Argument	Description
-trace	Generates a trace in the file defined in the grammar configuration file.
-indent	Generates the indented trees file.
-ne	Displays the number of the dependencies. Does not display the chunk tree.
-np	Add the phrase number to the dependencies. Does not display the

Argument	Description
	chunk tree.
-npi	Functions the same as the –np command, but also displays the inverted form of the dependencies.
-warning	Displays the unknown tag warnings for lexicon tags that have not been associated with a part of speech.

Refining the Grammar with Custom Definitions

You can refine the core grammar files described in the previous chapter using parameter files that contain custom rules that are appended to the core grammar. Parameter files can contain information such as feature definitions, category definitions, and deduction rules.

For example, parameter files are useful when your core grammar is encrypted to prevent modifications by the end user. You can also use parameter files to refine the core grammar for specific domains, such as defining subject/object dependencies.

The parameter file contains two fields at the top of the file: the boundaries field and the zones field. These fields are not required. However, when both fields are defined, the boundaries field must appear before the zones field.

Using the Boundaries Field

The boundaries field is the first field of the parameter file. It can accept more than one string defining a linguistic boundary. For example, the boundaries field can define the following strings as possible linguistic boundaries:

```
Boundaries:
</h1>
+SENT
</p>
```

The information in this field replaces and boundary definitions made in the controls file. For more information about the controls file, refer to "Defining Feature Defaults and Display Options" on page 19.

Using the Zones Field

The dependency rules can be defined in specific zones. These zones can be activated or deactivates using the zones field. Zones that are not active will not apply during parsing time.

The zones field lists the zones that should be activated. For example, the following zones field activates zones 1, 2, and 3:

```
ZONES:
1,2,3.
```

By default, the value of this field is an asterisk (*), meaning that all zones are active.

To use zones in the dependency rule, you must introduce the zones operator. The following rules belong to the zones defined until a new zones operator is introduced:

```
<zones:*>
//The rules below belong to all zones.
subject(#2,#1)=NP[fonc:~,fonc=subj]{?*,#1[last]},FV{?*,#2[last]}.
subject(#2,#1)=NP[fond:~,fond=subj]{?*,#1[last]},?*,FV{?*,#2[last]}.

zones:<1,2,3>
//The rules below belong to zones 1, 2, and 3.
object(#2,#1)=FV{?*,#2[last]},NP[fonc:~,fonc=subj]{?*,#1[last]}.
object(#2,#1)=FV{?*,#2[last]},?*,NP[fonc:~fonc=subj]{?*,#1[last]}.

<zones:1>
//The rule below belongs to zone 1.
if (subject(#2[!pers:!],#1) & object(#2,#3)) SVO(#1,#2,#3).
```

For more information about the zones operator, refer to "Operators in Dependency Rules" on page 52.

Using the –p option

You can introduce as many parameter files as you like using the –p option. If a file contains chunking rules, then the filename can start with a plus (+) to indicate that the layers of these rules should be appended to the core grammar. Otherwise, the merging of new rules will be made according to their absolute layer number.

For example, three parameter files are appended to the core grammar as follows:

```
echo "the lady sleeps" |parse –p +file1.txt –p file2.txt –p file3.txt
```

In the above command, the rules in the file1.txt file will be appended at the end of the chunking grammar, while the rules in the other files will be merged into the grammar according to their layer numbers.

Appendix A:
XIP API Reference

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

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. For more information about the parameter files, refer to "Refining the Grammar with Custom Definitions" on page 61.

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. This object is described in "Using the XipResult Class" on page 70. 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. This object is described in "Using the XipResult Class" on page 70.

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. For more information about hiding dependencies, refer to “Hiding Dependencies” on page 14.

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 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 B:
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

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

Element Name	Description
	the chunking rules). A NODE structure is built upon the division in TOKENs. 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 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: a tool that linguists can use to create finite-state networks, including the lexical transducers that do morphological analysis and generation.

Index

Z