Xerox Incremental Parser

User's Guide

06/01/2016

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

Audience

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

Conventions Used

This manual uses the following style conventions:

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

What This Manual Contains

This manual contains the following sections:

- Overview:
  Introduces how XIP parses text, including a look at the architecture of XIP and an overview of each XIP module.
- Basic Rule Concepts:
  Describes the principles for writing rules, including how data is structured, operations on categories and features, and the basic structure of rules.
- Writing Rules:
  Describes how to write the different types of XIP rules, including declaring features and categories and creating the four basic types of XIP rules.

- ♦ Implementing a XIP Grammar:
  Provides information on how to configure, initialize, and launch XIP.

Other XIP Documentation

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

Chapter 1: **Overview**

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

- XIP Architecture
- Introduction to a XIP Grammar

XIP Architecture

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

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

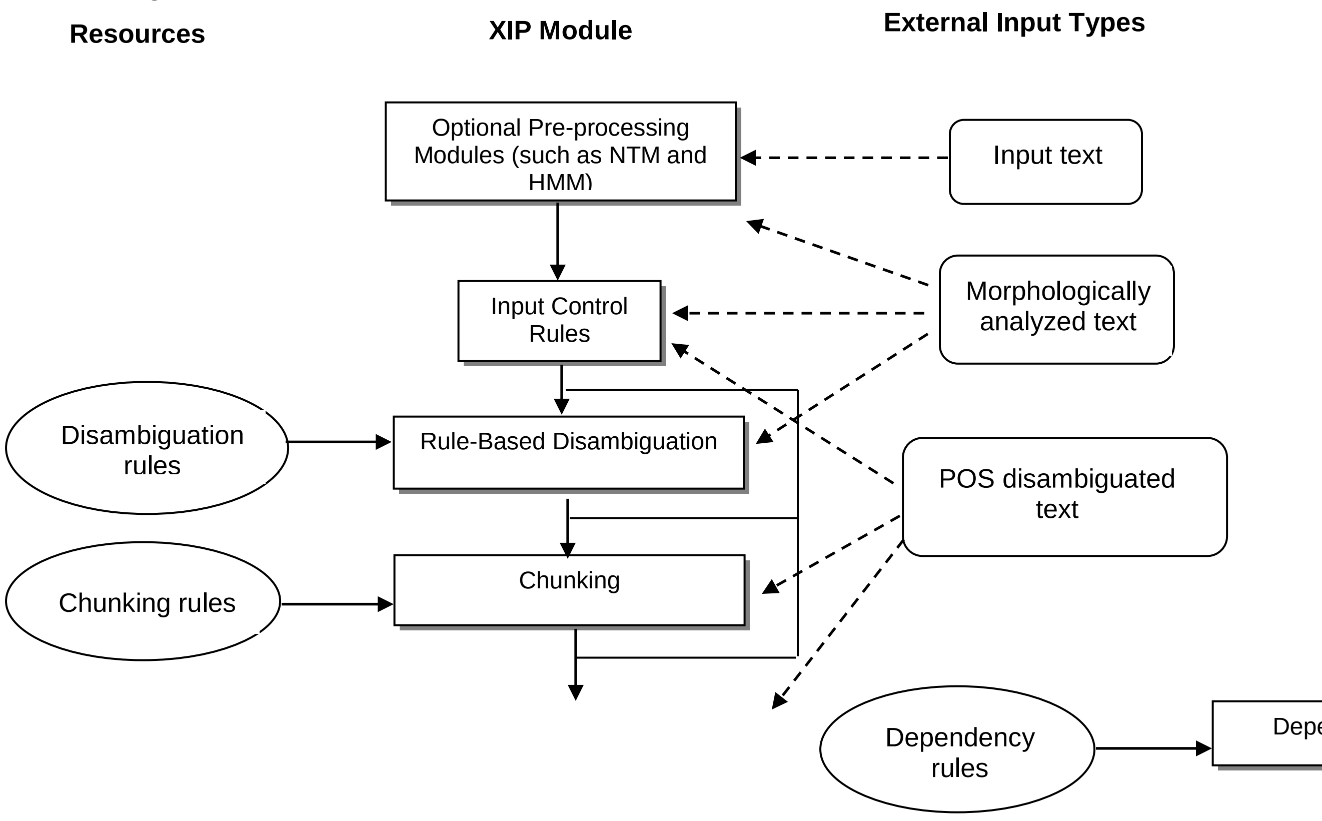
- Contextual disambiguation module: this module uses XIP rules to assign words features or categories according to their context.
- Chunking module: this module segments linguistic units as a series of chunks or other groupings.
- Dependency module: this module uses rules to identify dependencies between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

The optional pre-processing modules are:

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

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

Figure 1: Architecture of XIP



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

About the Input Data

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

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

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

About the Core XIP Linguistic Units

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

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

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

About the Input Control Rules

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

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

About Disambiguation

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

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

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

Can I help you? //verb
```

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

About Chunking

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

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

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

About Building Dependencies

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

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

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

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

```
Subject(eat, dog)

Object(eat, soup)
```

Introduction to a XIP Grammar

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

About the Grammar Files

A grammar contains the following basic types of files:

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

About Rules

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

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

Chapter 2: **Basic Rule Concepts**

This chapter includes the following sections:

Representing and Comparing Nodes

Data in XIP is represented as a sequence of nodes. An elementary lexical node represents a single lexical reading and consists of one category, one lemma, and a set of feature-value pairs. A terminal set describes a set of one or more lexical readings associated with a given lemma.

Following are examples of two nodes, Dog and chases:

```
Dog:noun[lemma:dog,surface:Dog,uppercase:+,sing:+].
chases:verb[lemma:chase,surface:chases,pres:+,person:3,sing:+].
```

Node expressions in rules match lexical readings based on criteria bearing on the category and features values. The following sections describe how sequences of nodes are represented in rules, the operators you can use to compare sequences of nodes, and the regular expressions you can use to compare the inner structures of nodes.

Nodes can be referred to in rules by their categories or by variables that are noted using #*n*, where *n* is an integer. For more information about using variables, refer to “Operations that Use Variables” on page 23.

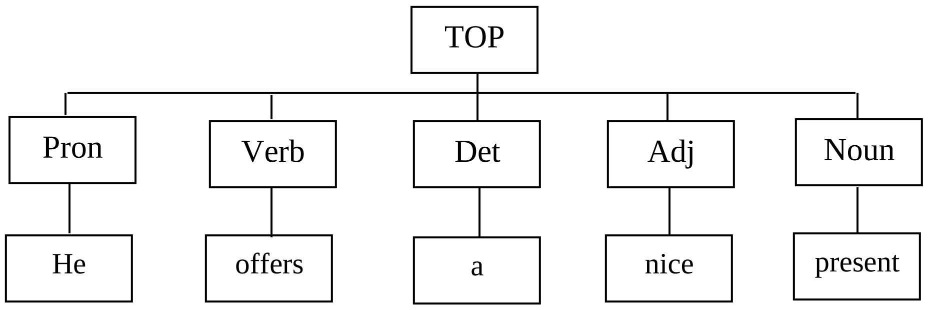
For information about the data itself and initializing the data structure, refer to “Initializing XIP” on page 46.

Representing Sequences of Nodes in the Chunk Tree

The chunk tree is made up of nodes. There are two types of nodes in the chunk tree, lexical nodes and non-lexical nodes. Lexical nodes describe the lexical units of the input text that make up the leaves of the chunk tree. Non-lexical nodes are used to unite a series of lexical nodes into chunks.

Figure 2 illustrates one possible chunk tree for the sentence “He offers a nice present”.

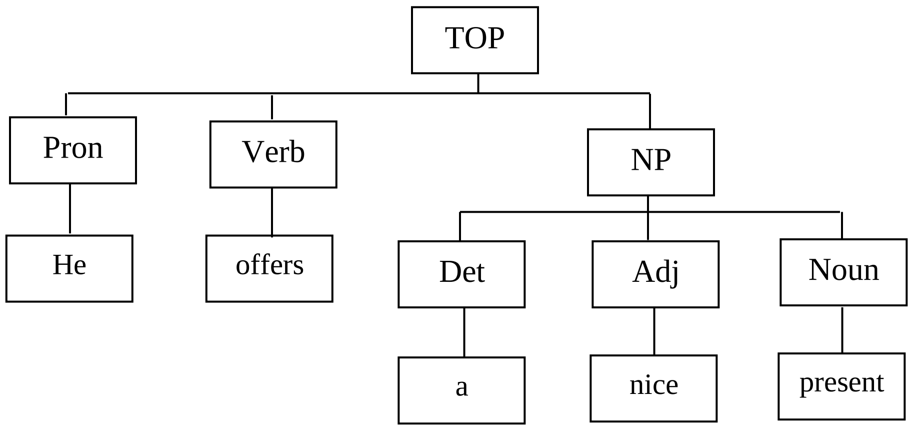
Figure 2: A Basic Chunk Tree



The root node in Figure 2 is labeled TOP. Each leaf lexical node is labeled with a category, such as Pron and Verb. The leaf nodes at the bottom of a chunk tree are called the terminal set. The terminal set describes a set of one or more lexical readings associated with a given token.

Figure 3 illustrates a more complex chunk tree for the same sentence. This chunk tree contains both lexical and non-lexical nodes.

Figure 3: A More Complex Chunk Tree



This chunk tree contains a Pron, Verb, and NP node. Under the NP non-lexical node, there are three constituent nodes: Det, Adj, and Noun. The Det, Adj, and Noun nodes are all sister nodes, meaning they share the same mother node, the NP node. The NP node is a sibling of the Verb and Pron nodes, meaning the nodes are related but do not share the same mother. The TOP node does not represent a mother node but serves as a placeholder only.

NOTE: A chunk tree can consist of a single chunk or node.

Operators for Defining Sequences of Nodes

Sequences of nodes are central to most rules. The following table describes the basic operators used in rules to define sequences of nodes.

Operator	Description
,	Concatenation. For example, det,adj specifies the det category followed by the adj category. This operator cannot be used in ID rules.
()	Optionality. For example, (adv) means that the adv category is optional.
*	Kleene star convention for naming zero or more instances. For example, adj* specifies zero or more adj categories.
+	Kleene plus convention for specifying one or more instances. For example, noun+ specifies one or more noun categories.
?	Any category.
;	Disjunction. For example, adv;adj means adv or adj.
{}	Sub-tree exploration. For example, NP{?*,noun} indicates that the sub-tree beneath the NP node will be searched for any sequence of nodes followed by a noun.
~	Specifies that a category cannot exist in a particular context. For example, ~noun means any category except a noun.

All of the operators can be combined, except for the tilde (~) operator. This operator cannot be applied to a set of elements.

Comparing Nodes

You can compare nodes using the operators described in the following table:

Operator	Description
::	Verifies if two nodes match. Consider the following example: #1::#2 This test succeeds if the node represented by variable #1 is the same as the node represented by variable #2.

Operator	Description
~:	Verifies that two nodes are different. Consider the following example: #1~:#2 This test succeeds if the nodes represented by variables #1 and #2 are different.
<	Verifies that one node precedes another in the chunk tree. Consider the following example: The lady opens the door. Assume that variable #1 represents the node lady and that variable #2 represents the node door. Then #1<#2 succeeds because lady (node #1) precedes door (node #2) in the chunk tree.
>	Verifies that one node follows another in the chunk tree. Consider the following example: The lady opens the door. Assume that variable #1 represents the node lady and that variable #2 represents the node door. Then #1<#2 fails because lady (node #1) precedes door (node #2) in the chunk tree.

Exploring the Inner Structure of Nodes

XIP provides a special type of regular expression called a tree regular expression (TRE) to establish connections between distant nodes. These expressions explore sub-nodes in depth. Braces ({}) are used in a rule to indicate that sub-nodes must be examined. An example TRE follows:

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

This expressions means that a noun phrase (NP) node that spans a determiner node and a noun node is followed, at the same level, by a finite verb (FV) node that spans a verb.

TREs can be used with variables and with the any symbol (?) and the disjunction symbol (;). Refer to "Operators for Defining Sequences of Nodes" on page 13 for more information about these operators.

Operations on Features and Categories

This section describes the different ways in which you can modify features and categories using the category definition file, rules, or a combination of both.

Performing Basic Operations on Features

Features and categories can be instantiated, created, and deleted. Values of features and categories can be tested.

Instantiating Features and Categories

Features and categories are instantiated using the equal sign (=) operator. For example, [gender=fem] sets the gender feature with the value fem. Instantiation fails if the feature is already set with a different value.

Testing Features and Categories

Features and categories can be tested using the operators described in the following table:

Operator	Description
[feature:value]	This feature and value must be present on the node for this feature or processing fails.
[feature:~]	The feature should not be instantiated on the node or processing fails. Note that negations are always tested before instantiation. For example, the following rule tests that a node has not been assigned a value and then assigns it a value: [fsubj:~, fsubj:+] The following rule is equivalent to the

Operator	Description
	previous, testing that a node has not been assigned a value and then assigning it a value, because negation is always tested first: [fsubj:+, fsubj:~]
[feature:~value]	The feature should not have the value specified.
[feature]	The feature should be instantiated. If not, processing fails.

Deleting Features and

You can delete features and categories from the node using the tilde (~) operator. For example, [acc=~] deletes all values of the acc feature.

Using Operators to Compare Features of Nodes

You can compare the values of features on a node using the following basic pattern:

```
#1[feature]:#2[feature]
```

#1 and #2 are two variables that represent nodes. For more information about variables, refer to “Operations that Use Variables” on page 23.

The following table describes the different operators you can use to compare features:

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

Operator	Description
	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: #1[agreement]::#2[agreement] This comparison succeeds if node #2 bears the same values for the agreement features 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 num of #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.

Using Boolean Expressions to Compare Features

You can use Boolean expressions, such as the “or” operator (|) or the “and” operator (&), to combine tests on features of specific nodes.

For example, the following expression determines if the gender feature of node #1 is feminine or masculine:

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

Boolean expressions are always evaluated from left to right and the operators are not prioritized. For example, the following two expressions are equivalent:

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

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

Parentheses can be used in a Boolean expression to constrain how it is interpreted.

Advanced Operations on Features

This section describes advanced methods for modifying and using features, including sharing features between daughter nodes and mother nodes (percolation) and using a special type of rule called V-rules.

Making Features Free or Bound

You can specify the status of features in the category definition or in the rules. Features can be either *free* or *bound*.

The values of free features are shared between the daughter node and its mother node. This inverse application of features (from the bottom of the chunk tree up) is referred to as 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.

A free feature is specified using the operators described in the following table:

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

For example, in the following category declaration the verbcat feature is set to + and is free in all VERBS. The transitive feature is free no matter what its value in all VERBS:

```
VERB=[!verbcat=+,!transitive:!].
```

The values of bound features do not percolate from a daughter node to its mother node. Bound features are indicated as follows:

```
[feature:!].
```

For example, in the following category declaration the transitive feature is bound in all FVs:

```
FV=[transitive:!].
```

Feature status (free or bound) can also be enforced in rules. For example, the following rule frees the animate feature in all noun phrases:

```
PP = Prep, NP[!animate:!].
```

Modifying Features Conditionally Using V-Rules

Valence rules (V-rules) provide a way to modify specific features when a condition is verified. There are two types of V-rules: default feature 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. Pre V-rules are also applied on the lexical nodes before any other kind of rules are applied.

Post V-rules are applied after the instantiation of free features from the daughter nodes to the mother node.

When an empty node is first created, the V-rules are applied as follows:

1. The pre V-rules are applied.
2. Then, free feature percolation is applied.
3. Finally, the post V-rules are applied.

Applying the V-rules before feature percolation allows you to correct any problems that may be caused by the percolation itself. Post V-rules are not involved in the percolation, thus they can contain tests that would have caused problems if they were percolated.

Using DFS Rules

The DFS rules append features to a node on the fly when a condition is verified. Each DFS rule is composed of two lists of features. The list on the left is the condition, and the list on the right is the set of features to be instantiated on the node if the conditions are met. Failure to instantiate these features does not cause the rule analysis to fail.

Following is an example of a DFS rule:

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

The rule adds the nominative feature to a node when the node contains the pron and indefinite features.

Using FCR Rules

FCR rules are used to check the consistency of a set of features on a node. The form of these rules is the same as the DFS rules, but their application follows a completely different pattern. If the left part of the rule applies, then the right part of the rule must also apply. Thus, failure to apply a FCR rule causes the rule analysis to fail.

Using V-Rules to Create Start and End Features

You can create start and end features using V-rules. The start and end features are automatically instantiated on the first and last nodes of the sentence. These features are used like other features. They can be used locally to verify the position of a node in the sequence. They can also be transformed into free features that percolate until the end of chunking.

For example, to create start and end features that percolate, use a PreDFS rule as follows:

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

About the Basic Structure of Rules

This section describes the basic structure of rules, including the files they are stored in, associating rules with a context, and implementing rules in layers.

About the Rule Files

Rules are stored in text files that, by default, have the extension .xip. We recommend separating your rules into different files for ease of maintenance and management. For example, the chunking rules could be stored in a file called chunker.xip and the dependency rules could be stored in a file called dependency.xip.

The disambiguation rules and the chunking rules can be stored in the same file or files because they are distributed over layers. A layer contains only one sort of rules. Layers are described later in this section on page 23.

Associating Rules With a Context

A rule of any type can be associated with a context that restricts its application according to sequences of categories on the left or right side of the selected nodes. A context is defined as a sequence of sub-nodes.

A context is always written between pipes (||). For example, consider the following rules:

```
2>NP->|?[noun:~]| AP[first:+], noun[last:+,proper:~].
```

The above rule means that an NP chunk is built from an AP followed by a noun if the category on the left of the AP is not a noun.

A context can be negated with a tilde (~) before the first pipe (|). For example, the following rule applies to any context except for the context that contains a noun followed by one or more adverbs:

```
2>NP->~|noun,adv*| AP[first:+], noun[last:+,proper:~].
```

Ordering Rules in Layers

During the execution of disambiguation and chunking rules, the processing stream is incrementally updated through ordered layers of rules. After each layer of rules has been applied, the processing stream is represented as a chunk tree under a virtual root node.

You can use layers to specify where in the process the rules are evaluated. Each layer is written as a number, between one and 300, followed by the greater-than sign (>). For example, layer nine is indicated in a rule as follows:

```
9> VP=pro,verb.
```

Layers are processed according to their rank, from the first layer (1) to the last. Each layer can contain only one kind of rule, such as disambiguation rules, sequence rules, or ID rules. You can leave empty layers to make space for rules that will be added later.

Rules are stored in files where layer numbers can be absolute or relative to the previous rule file. Layer numbers can be absolute, meaning determined by a number written in a file, or relative, meaning they are calculated depending upon the rules files listed before or after them in the grammar configuration file. For more information about listing rules in the grammar configuration file, refer to "Configuring XIP" on page 42.

The system of using relative layer numbers allows you to incorporate new rule files easily. If a core grammar has been written that needs to be further refined with more chunking rules, those rules can be stored in a new file in which the first layer starts at one. The new layers will be appended after the layers stored in the previous file.

For example, file A contains layers 1-15 and file B contains layers 1-5. If file B is treated as relative (specified with a plus (+) before its name in the grammar configuration file), then layer 1 in file B actually corresponds to layer 16.

Operations that Use Variables

In all rules, you can associate a node with a variable to test specific values or to compare nodes on the basis of features.

Syntax of Variables

Nodes can be associated with a variable of the following form:

```
#number
```

The *number* must be greater than zero.

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.

You can add comments to variables that help improve the readability of complex rules. For example, you might assign comments to two variables, #1 and #2, as follows:

```
if(subj(#1_verb, #2_noun)
```

The comments help you remember that variable #1 is assigned to a verb node and that variable #2 is assigned to a noun node. However, the comment itself has no effect on how the rule itself.

Using Variables to Copy Features to Distant Nodes

You can use variables to copy features between nodes that are distant. To do so, combine the equal sign operator (=) with the free feature declaration. For more information about declaring free features refer to “Advanced Operations on Features” on page 19. For more information about representing nodes, refer to “Representing and Comparing Nodes” on page 11.

To copy features between two nodes you need to first define two variables and then define in the right part of the equation the features to be copied.

In the following example, the feature gender is copied from node #2 to node #1:

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

Node #2 can be further constrained with specific features as follows:

```
#1[gender]={#2[number:plur]}.
```

In the above example, the feature is copied only when node #2 is plural. No feature can be declared in the left part of the equation. If copying features fails, then instantiation fails.

Using the “where” Keyword with Variables

You can use the where keyword in combination with variables to specify constraints on features across different nodes in a given rule as follows:

```
Det#1[first],(Ap),noun#2[last,proper:~], where(#1[gender]::#2[gender]).
```

The above rule applies only if the Det and the noun nodes have the same gender.

The where keyword can also be used to assign features to selected nodes. For example, the following rule assigned the gender feature to the Det and noun nodes:

```
2>NP-> Det#1[first], (Ap), noun#2[last,proper:~],
    where(#0[gender]={#1 & #2}).
```

The following rule gives the new gender feature a value:

```
2>NP-> Det#1[first], (Ap), noun#2[last,proper:~],
    where(#0[gender=fem]).
```

By default, the variable #0 refers to a node that is being created on the left side of the rule.

Using Operators with Variable

You can use all of the usual operators with variables. One additional operator, ::, imposes a strict comparison between features. This operator requires all fields to be instantiated in both structures. For example, the following rule imposes a strict comparison between agreement features:

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

This comparison succeeds if node #2 bears the same values for the agreement features as node #1.

Chapter 3:
Writing Rules

This chapter describes how to write the different types of rules used in XIP grammars. It includes the following sections:

Declaring Features, Categories, and Dependencies

XIP, like programming languages, requires you to declare the feature, category, and dependency names used by the rules. The following sections describe how to declare each type of name.

Declaring Features

Each node in the chunk tree is associated with features. A feature expresses properties of the node, such as singular or plural, number, and gender. A feature is instantiated as a feature:value pair. Rules may also instantiate new features on a node.

Features are listed in the feature declaration file (by default, features.xip). Features can also be provided in a custom lexicon file.

Syntax

Features are declared using the following syntax:

```
feature1:{value1,value2,...,valueN},
feature2:{value1,value2,...,valueN},
...
featureN:{value1,value2,...,valueN},
```

In the above syntax, feature1, feature2, and featureN are the feature names and value1, value2, and valueN represent the collection of different values available for the feature. Lists of values are always defined using braces ({}). A list of values is known as a domain of features.

The values can be any string, integer, or range of integers (such as 1-3).

NOTE: The same node cannot have two features that have the same name with different values. For example, the same node cannot have the feature feature1:value1 and the feature feature1:value2.

Defining General Features

XIP allows you to define general features that refer to a group of features. General features are used to evaluate if a node in the chunk tree contains one of the features referenced by the general feature. A list of features is defined using square brackets ([]).

For example, to describe the participle forms of verb, we define two features: paspart:+ for the past participle and prespart:+ for the present participle. Next, we can create the general feature, participle that has two verb forms, present participle or a past participle. The general participle feature refers to the set of features for the paspart and prespart features. The general feature is declared as follows:

```
participle:[
  paspart:{+},
  prespart:{+}]
```

An actual feature declaration file may contain many features and general features, as follows:

```
[Features:
  [dir:{+}
   indir:{+},
   agreement:[gender:{fem,masc,neut},
              number:{sing,plur,dual},
              case:{nom,acc,gen,dat,loc}],
    pers:{1-3},
    participle:[
               paspart:{+},
               prespart:{+}]
  ]
]
```

Declaring Categories

Features permit the specification of categories. A category is a collection of features. For example, a category might consist of the following three features:

```
noun: +
gender: fem
number: pl
```

The part of speech is the value of one of the features contained by this category, in this case the noun feature. Lexical categories are categories associated with a part of speech value. Non-lexical categories are not associated with a part of speech.

When a category is used in a rule, the default features (those specified in the definition of the category) are always appended to the matching element in the rule. Categories are declared with at least one feature.

Syntax

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

```
category_name =[feature1=value1,feature2=value2].
```

The category name is given on the left of the equal sign (=) and its set of features is given on the right of the equal sign. A list of features is composed of a feature tag, such as nominal, and its value, such as +.

Creating the Virtual Root Node

XIP builds chunk trees from parsed text. The root of these chunk trees is not computed by the rules. Instead, a virtual root node is created that spans the sequence of nodes, offering a more consistent structure on which to apply rules.

The first category defined in the category declaration file is always used as the virtual root node. For example, consider the following category definitions:

```
top = [top=+]
noun = [nominal=+,verbal=-,bar=1].
verb = [verbal=+,nominal=-,bar=1].
vp = [nominal=-,verbal=+,bar=2].
```

The category used for the virtual root node of the chunk tree is top because it is the first category defined in the file.

Declaring Dependencies

XIP may define dependencies between nodes in the chunk tree. All dependency tags used by the dependency rules must be declared. A dependency is declared in the dependency declaration file as follows:

```
Functions:
   subj.
   obj.
   vmod.
   neighbor.
```

Note that you must use a period after each dependency tag declaration.

Some dependencies are used for computing reasons only or should not be present in the final display. These dependencies do not need to be deleted, only hidden. To hide a dependency, it must first be declared after the Functions field. Then, you hide it by declaring it after the Hidden field as follows:

```
Hidden:
   neighbor.
```

NOTE: When using the XIP API, the dependencies declared as hidden are not stored in the XipDependency object.

For more information about the XIP API, refer to the *XIP Reference Guide*.

Dependencies can also bear features of their own. Features of dependencies must be declared in the feature declaration file and the controls file, which controls how dependencies are displayed. Refer to "Defining General Features" on page 27 for more information about declaring features. For more information about displaying the dependency features, refer to "Displaying XIP Results" on page 49.

Writing Input Control Rules

The input control rules pre-treat the input text before it is processed by the other XIP modules. Input control rules consist of two types of rules: lexical rules and split rules.

Using Lexical Rules

Lexical rules enrich or replace tokens provided by NTM or other external lexical pre-processing services, such as tokenization services or lexical lookup services.

Lexical rules are stored in internal lexical files. These lexical files append new features to existing categories, modify or replace categories, and add new categories for existing words. The use of lexical rules is optional.

The lexical rules assign features to unique lemmas. 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.

The first line in the file that contains the lexical rules must contain the following keyword:

```
Vocabulary:
```

This line tells the XIP engine that the file contains a custom lexicon. All categories and features in lexical rules must be declared in the corresponding declaration files. For more information about declaring categories and features, refer to “Declaring Features, Categories, and Dependencies” on page 26.

Using Split Rules

After the lexical rules have been applied, the split rules break the input stream into elementary processing units (usually a sentence). A processing unit is represented in XIP as a sequence of nodes. A split rule is defined as a sequence of nodes, where the last node defines the breaking point. Split rules are processed sequentially. You can specify more than one split rule.

For example, the following split rule breaks the input after a colon (:) when a verb followed by no other punctuation is found on the left side of the colon:

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

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

```
|SENT|
```

Writing Disambiguation Rules

The external lexicon often yields several lexical readings for a given token. The role of the disambiguation rules is to determine which of these readings are kept or disallowed in a given context. Readings are categories and their features.

For a set of readings, the disambiguation rule selects a subset of readings depending upon the token’s surrounding context. In addition, disambiguation rules can assign and modify feature-value pairs for selected readings.

Syntax of a Basic Disambiguation Rule

The syntax of a basic disambiguation rule follows:

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

The following table describes the different parts of the rule:

Variable	**Description**
layer	A layer number. Layers allow you to alternate disambiguation rules with other sorts of rules, such as sequence rules. A rule with no layer number is automatically inserted in layer zero (0), the layer applied before any other rules of any kind are applied. For more information, refer to “Ordering Rules in Layers” on page 23.

Variable	Description
readings_filter	An expression that specifies a subset of categories and features associated with a token. The disambiguation rule applies to any token whose lexical readings are a subset of the *readings_filter*. For example, an ambiguity class of noun,verb indicates that the word is either a noun or a verb.
left_context *right_context*	A sequence of nodes (optional). For more information about contexts, refer to “Associating Rules With a Context” on page 22.
selected_readings	The categories and features to be selected from among the lexical readings associated with the tokens found by the *readings_filter*. If a rule pattern matches some segment in the current input stream, the terminal set is updated. Only readings that match the *selected_readings* are kept.

Operators and Regular Expressions in Disambiguation Rules

Disambiguation rules can use the usual operators and regular expressions to specify conditions on features. For example, a left context might contain the following:

```
|noun[thatcomp:+,verb:~],?[conj:~],adj;adv|.
```

This specifies a left context that contains a noun that bears the thatcomp feature and does not bear the verb feature, followed by any word that is not a conjunction, followed by an adjective or adverb.

You can use angle brackets (<>) to define specific features associated with a reading. For example, the following expression refers to a token that has a singular noun and a verb reading:

```
noun<sing:+>, verb
```

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

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

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

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

Following is another example of an expression that uses brackets:

```
(noun,adj)[present:~]
```

This expression indicates that a token can be both a noun and an adjective, but can bear no present features.

For more information about using contexts in rules, refer to "Associating Rules With a Context" on page 22.

You can use the percent sign (%) operator to enforce the selection of lexical readings for nodes mentioned in the left or right context The % operator can also be used to replace the terminal set by a new lexical 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=+].
```

Writing Chunking Rules

The chunking module takes as input a list of lexical nodes, each built upon a word node. These nodes bear features that originate from the lexicon, the disambiguation rules, and the V-rules.

The chunking rules group sequences of categories into structures that can be processed by the dependency module. These structures are called chunk trees. Refer to "Disambiguation rules can also be used to redefine feature values based on the context of the node.

About Chunking" on page 9 for more information about chunking and the chunk tree.

The central chunking module is composed of two types of rules:

- Immediate dependency and linear precedence rules (ID/LP rules)
- Sequence rules

After these rules have been used to build a chunk tree, you can apply rules that modify the chunks. Refer to "Modifying Chunks Using Marking and Reshuffling Rules" on page 40 for more information.

This section describes how chunking rules are executed and how to formulate each type of rule.

About Chunking Rule Execution

Chunking rules are organized in layers. The layer is written as a number, between one and 300, followed by the greater-than sign (>). Layers are processed sequentially. For more information about layers, refer to "Ordering Rules in Layers" on page 23.

The application of a chunking rule is definitive. Once a rule has been applied, the resulting chunks are never dismissed and are passed to the next layer. The chunk tree is updated accordingly.

ID/LP rules and sequence rules cannot belong to the same layer. The LP rules must be present in the same layer as the ID rules they restrict.

Creating ID/LP Rules

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

About ID Rules

An ID rule has the following syntax:

```
layer> new_node -> list_of_lexical_nodes.
```

If an ID rule applies to an unordered set of nodes (defined in the *list_of_lexical_nodes*), then a new node is created (*new_node*) and the chunk tree is updated accordingly. The new node is available for subsequent chunking rules. The initial nodes are no longer available.

You can use optional categories (specified by parentheses) and the Kleene star convention in ID rules. For more information about these conventions, refer to "Representing and Comparing Nodes" on page

11.. ID rules can also use the where keyword, which is described in "Operations that Use Variables" on page 23.

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

For example, the following ID rule states that a determiner is optional and that the rule accepts zero or more adjectives:

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

About LP Rules

ID rules define a set of categories, and the LP rules impose some conditions on this order.

LP rules are defined as conditions on features rather than on the categories themselves. LP rules can be defined in a layer, and will be associated with all of the ID rules belonging to the same layer. You can also create general LP rules that do not have a layer number, which will be used as a general constraint throughout the rules.

LP rules have the following syntax:

```
layer> [set of features] < [set of features].
```

The layer number is optional. For example, the following LP rule means that every adjective precedes a noun in all ID rules:

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

You can control the order of categories using LP rules or using the first and last features, which are always instantiated on the first and last categories of a phrase. For example, the following ID rule introduces a constraint on the position of the determiner as the first category of the phrase and the noun as the last category:

```
NP->det[first],adj,noun[last].
```

Algorithm of ID/LP Rules

When applying ID/LP rules in a given layer, first the longest possible sequence of valid nodes is isolated in the input stream. A valid node is any node that belongs to the right side of a rule in an active layer. Next, the rules from the layer are tested against this sequence. The longest sequence from right to left determines the rule applied in a given layer. When more than one sequence competes for longest match, the first rule in the layer applies.

Creating Sequence Rules

Unlike ID/LP rules, which work on an unordered group of nodes, sequence rules describe an ordered sequence of nodes. The rules apply sequentially in a given layer according to the order you define. The input stream is scanned from left to right until the whole input stream has been processed.

Syntax

Each sequence rule applies from left to right using the equal sign (=) operator or from right to left using the <= operator.

Sequence rules have the following syntax:

```
layer> new_node = list_of_lexical_nodes.
or
layer> new_node <= list_of_lexical_nodes.
```

Sequence rules can use the basic sequence operators as described in "Operators for Defining Sequences of Nodes" on page 13. Sequence rules can also use the where keyword, which is described in "Operations that Use Variables" on page 23.

Sequence Rule Algorithm

During sequence rule execution, the input stream is traversed until a node that bears a valid initial category is found. A valid initial category is a category that starts a sequence rule in a given layer. For example, det is the only valid initial category in the following layer:

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

Next, sequence rules that start with the category of the valid initial node are tested one after the other, starting at the valid node. The first rule to match a sequence is selected and the input stream is updated accordingly.

Finding the Shortest Match Versus the Longest Match

A sequence rule applies to the shortest match when you use the = or <= operators. For example, the following sequence rule applies from left to right and identifies the shortest match:

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

You can apply a 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.
```

Indexing the Lemma of a Sequence

Sequence rules can be indexed on the lemma of the first or last node in the sequence. This index provides an efficient way to define lexical rules, such as rules for describing multi-word expressions. An indexed sequence rule is triggered when a word matching the index is present in a text.

The index is isolated just after the layer number and followed by a colon. For example, the following sequence rule is indexed on the word “as,” so can be used to describe the multi-word expression “as long as”:

```
6> as: CONJ = prep[start], adj[lemma:long], prep[form:f_as].
```

Building Dependency Relationships

Dependency relationships connect nodes according to specific relationships, typically standard syntactic dependencies, but also broader relationships, including relationships across sentences.

Dependency rules take as input a sequence of constituent or lexical nodes, such as those produced by the chunking module and can create the following types of dependency relationships:

- Create a new relationship between nodes.
- Assign new features to a node in the chunk tree.
- Delete an existing relationship.
- Rename an existing dependency relationship.

More than one dependency can be defined per rule. The rules can be written as a single test rather than based on a pattern. You can also create rules that do not reference the chunk tree.

Format and Syntax of Dependency Rules

Dependency rules are composed of three parts:

- A regular expression pattern.

- A collection of conditions about relationships between the nodes of a chunk tree or the nodes themselves, independent of the tree structure.
- A dependency term.

The syntax of dependency rules consists of the regular expression between two pipes (|), the key word if, which introduces the conditions in parentheses, followed by the conclusion. Dependency rules are processed sequentially.

The syntax of a dependency rule follows:

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

The *pattern* contains a tree regular expression (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. You can replace the if keyword with the where keyword to evaluate the expression in a deterministic way.

For more information about TREs, refer to "Exploring the Inner Structure of Nodes" on page 15. For more information about Boolean expressions, refer to "Using Boolean Expressions to Compare Features" on page 18.

The *dependency_term* is the new dependency created by the rule. It has the following form:

```
label[features](arg1,arg2,...,argn)
```

The *label* is the name of the dependency and *arg1*, *arg2*,...,*argn* are the arguments of the dependency relations. There can be any number of arguments. The *features* expression defines how feature-value pairs can be associated with the dependency relation. The *features* expression is optional.

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 Dependency Relationships

The following sections describe the different types of modifications that can be made to dependencies.

Adding Features to Nodes

Dependency rules can bear features that are used to refine the meaning of the dependency relationship. The feature can be appended to the dependency name identified. For example, consider the following dependency rules:

```
|NP{?*,#1[last]},VP{?*,#2[last]}|   Subj(#2,#1).
|NP{?*,#1[last]}, VP[passive]{?*,#2[last]}|  Subj[passive=+](#2,#1).
```

The first rule builds a subject dependency between a noun phrase (NP) and a verb phrase (VP). The second rule appends the passive feature to the subject dependency when the verb phrase bears a passive feature.

Renaming Dependencies

You can rename dependencies depending upon new dependency relationships. For example, the following dependency rule renames the dependency. The carat (^) symbol marks the dependency modified by the rule.

```
if (^vmod(#1,#2)&prep(#3,#2)&#1[fsubcat]:#3[fsubcat]) varg(#1,#2).
```

The above dependency rule changes the vmod dependency to varg if the subcat feature of the vmod dependency is compatible with the subcat feature of the prepositional phrase

Deleting Dependencies

You can delete dependencies using dependency rules. For example, the following dependency rule eliminates the right subject if the left subject is available:

```
if (subj[left](#1,#2) & ^subj[right](#1,#3)) ~.
```

Again, the carat (^) marks the dependency modified by the rule.

Advanced Rules

This section describes features of rules available to advanced XIP grammar writers.

Advanced Sequence Rules

You can modify sequence rules so that they add or remove new features on specific nodes. These special sequence rules are composed like regular sequence rules except the left part of the rule contains the any symbol, a question mark (?). No mother node is built on top of the sequence of nodes identified.

For example, the following sequence rule instantiates the det feature on the noun node:

```
? = |det| noun[det=+].
```

You can associate features with the question mark on the left side of the rule. These features are then instantiated on all nodes that match the rule. For example, if the following rule applies, then the adj and noun nodes receive the nominal feature:

```
?[nominal=+] = |det| adj, noun.
```

Modifying Chunks Using Marking and Reshuffling Rules

Once you have constructed chunks using the chunking rules (ID/LP rules and sequence rules), you can modify the chunks and the chunk tree using marking rules and reshuffling rules.

For more information about chunking rules, refer to "Writing Chunking Rules" on page 33.

About Marking Rules

Marking rules mark specific node configurations in the chunk tree. You can use marking rules to explore sequences of sub-tree, for example to mark a chunk in the chunk tree as passive. Marking rules must be defined within specific layers. All nodes, at any level, can be marked.

Marking rules use tree regular expressions (TRE), which extract specific relations between distant nodes. For more information about TRE, refer to "Exploring the Inner Structure of Nodes" on page 15.

For example, the following marking rule marks a FV node as passive when the auxiliary is "be" and the next node is a prepositional phrase introduced by the preposition "by":

```
15> FV[passive=+]{Vaux[aux:be],Verb[pastpart]},PP{Prep[by]}.
```

About Reshuffling Rules

Reshuffling rules rebuild sections of the chunk tree. You can dispatch reshuffling rules across layers, like other chunking rules, or alternate them with other chunking rules.

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. Each node in the left of a reshuffling rule is associated with a variable. The right side of the rule defines a new, reshuffled sub-tree based on the variables.

Unlike other rules, a variable in a reshuffling rule can represent a list of nodes.

An example reshuffling rule follows:

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

The above 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 as follows:

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

Chapter 4:
Implementing a XIP Grammar

This chapter describes how to implement your custom XIP grammar. It contains the following sections:

- Configuring XIP
- Initializing XIP
- Launching XIP Using the Command Line

Configuring XIP

Each tag used by the rules of the XIP grammar must be declared. The remainder of this section describes how to create a grammar configuration file and how to declare the files that contain the definitions of the features, categories, and dependencies used by XIP.

General File Conventions

The following conventions apply to most types of XIP files, including the files used to configure the system.

Writing Comments

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
```

Escaping Characters

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

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

Creating a Grammar Configuration File

The grammar configuration file defines all of the files that compose your grammar, such as the rules files, lexicons, and declaration files. By default, the grammar configuration file is given the name of the language of the grammar, such as english.xip.

The following table describes each field of the grammar configuration file:

Field Name	Description
License	Contains the name of license holder.
Modules	Describes the XIP modules for which the license is valid.
Expiration	Provides the license expiration date, in year/month/day format.
Code	Contains a key computed from the License, Modules, and Expiration fields. This checking key confirms the system was installed correctly.
Language	Gives the default language of your system. The language you provide here determines the language of the field names in your grammar files. The field names are provided by strings.file for English and chaines.fic for French. By default, the Language field is set to French. To add a new language, you need to modify one of the string files. If the string file is not present in your XIP directory, it will be recreated automatically as you run XIP.
Locale	Specifies the locale of the grammar. This field was used in previous version of XIP to localize the codeset. This field is no longer in use.

Field Name	Description
Number	Tells the system the maximum number of terminal features (those specified using {}) present at compilation time. Each feature value is replaced by a binary (bit). The Number field specifies the number of integers used to store the field values. Because an integer stores 64 values, to store a maximum of 320 terminal features you would set this field to 5 (5 x 64). To store more terminal features, increase this number.
Indentation	Gives the name of the trace file that describes the chunk tree using indentations. One space is added for each node level. This file is useful for displaying nodes along with their features. By default, the name of this file is trees.out.
Trace	Gives the name of the trace file that contains information about the execution of the XIP application. By default, the name of this file is trace.out.
Features	Contains the name of all the base files that define the features and categories used by your grammar. The features, categories, and translations used by NTM can be all contained by one file or split across many files. By default, this fields lists the following features files: features.xip,categories.xip,translations.xip, controls.xip,functions.xip

Field Name	**Description**
Lexicons	Provides the name of the internal lexicon file(s). The lexicon file contains supplementary words, new meanings for existing words, or a single meaning that is imposed on a word. You can also declare translations for HTML tags in this file. By default, the name of this file is lexicon.xip. For more information about creating internal lexicons, refer to "Using Lexical Rules" on page 30.
Rules	Lists the rules files applied to the linguistic input. XIP applies rules according to their order in this field. Different types of rules can be split into different files, such as dependency.xip for the dependency rules and chunker.xip for the chunking rules. You can use a plus (+) in front of the file name to indicate that the layer numbers in this file are relative. For more information about layers, refer to "Ordering Rules in Layers" on page 23. For example, this field contains the values adjust.xip,+localgram.xip. The first file listed, adjust.xip, is not preceded by a plus, so its layer numbers are absolute. The second file, +localgram.xip, has a plus, so its layer numbers are computed using the last layer number in the adjust.xip file added to each layer number in the localgram.xip file.

Initializing XIP

XIP initial data structures can be instantiated by the following:

- A lexical lookup process, such as Xerox FST input provided by NTM.
- A XIP XML standard input stream.

The following sections describe standard input data, external input data, and initializing XIP data structures using the two processes listed above.

About the Standard Input Data

The input stream is split into core processing units that represent, for example, sentences or paragraphs. The boundaries of the core processing units are defined by selected sequences of nodes in the input stream, for example end of phrase markers, such as a period (.), colon (:), or semi-colon (;).

The initial processing unit is represented as a sequence of terminal sets. A terminal set, as described earlier, is a set of one or more nodes associated with a given token. A node represents a lexical reading with a category, a lemma, and a set of features and values.

You can use whatever processing unit delimiters you want. For example, if you want to process paragraphs of HTML input, you can use the HTML tags <P> and </P> to delimit your processing units.

Using External Input

You can use the output of other tokenization and morphological analyzers as input for the XIP disambiguation module. However, the output of an external process may vary depending on the lexicons and syntactic tags being used.

XIP provides a way to keep the grammar independent from these external tags. The translation file provides a simple way to translate external tags to XIP tags so that there is no need to change the grammar when external input is used. Translation rules have the following syntax:

```
external_lexicon_tag = (XIP_tag)([features]).
```

The *external_lexicon_tag* is the part of speech or feature from the external lexicon. The *XIP_tag* is the translation to the XIP equivalent tag. Every rule must end with a period (.).

Following are some example translation rules:

```
Nom = noun. //translates a Nom tag to a XIP noun
FEM = [gender=fem]. //translates FEM tag to a feature and list of features
DAY = noun[time=+,day=+] //translates DAY tag to a POS with a feature set
```

If your input data contains tags that you have not translated, XIP will skip the unknown tag and continue parsing. You can use the –Warning option in the command line to have a list of the unknown tags. They will be stored in the errors.ers file.

Using Lexical Lookup

You can use any pre- process for tokenization and lexical lookup to instantiate the XIP initial data structure according to the following format:

```
word_form \t lemma \t features
```

NTM is an example of a lexical lookup process that takes the standard output of the Xerox FST (xfst) lexicons and converts it.

Using XIP XML

You can also use the XIP XML standard input stream to instantiate the initial XIP data structure. This input stream is based on the XIP DTD, and consists of a sequence of morphologically analyzed tokens and chunks. The XIP XML DTD defines a chunk tree and dependency tree using a series of XML tags.

To produce XIP XML standard data, use the –xml command in the command line. Refer to "Displaying XIP Results" on page 49 for more information about using the command line. For more information about the format of the XIP XML DTD, refer to Appendix B of the *XIP Reference Guide*.

When using XML input, you must identify how XIP interprets the XML tags. Certain characters, such as a quote ("), ampersand (&) or angle bracket (<), are forbidden in XML fields and must be replaced with a special encoding. The XML coding file specifies the XML coding chosen for specific characters. If the characters cannot be displayed, their internal code can be used instead. The internal code is then preceded with a number sign (#). Examples of XML coding file entries follow:

```
XMLCoding:
  " = &#34;
  < = &#60;
```

Launching XIP Using the Command Line

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

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

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

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

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

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

Displaying XIP Results

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

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

Argument	Description
-a	Displays all of the nodes.
-r	Displays a reduced set.
-tr	Displays even less node information.
-lem	Displays the same node set as the –tr

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

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