Xerox Incremental Parser

User’s Guide (Scripting)

6, CH DE MAUPERTUIS
38240 MEYLAN
France

6, CH DE MAUPERTUIS
38240 MEYLAN
France

Contents

Addenda

This document describes the new features that have been appended to XIP since the last version of the manual.
The modifications consist of new sorts of rules and new commands that can be used with the last version of the XIP engine.

Charset management: Language file commands

Charset encoding is a source of many issues when a document must be parsed:

a) The input could be in UTF8 or a different encoding
b) The internal lexicon or the grammar could be in UTF8 or a different encoding
c) The FST could be encoded in UTF8 or a different encoding

Each of these three situations can be covered in XIP in the following manner:

Command line options

First, XIP offers two command line options:

a) inpututf8
b) testinputf8

The first command will indicate to the parser that the input is in UTF8, the second command will test whether the input is in UTF8 and takes appropriate measures.

fstcharset

fstcharset is a command that should be added to the language file. It indicates whether the FST has been encoded in UTF8 or in latin. In this case, the system will convert the input into UTF8 if the command line options above are missing. It accepts either *utf8* or *latin* as a value.

```
fstcharset: utf8
```

grammarcharset

This command indicates that the whole grammar is in UTF8 or in latin. In that case the input will be converted into UTF8 to match the grammar. The default value is *latin*.

```
grammarcharset: latin
```

convert_lexicon_to_utf8_from_latin

This command is the most complicate of all. Its value is one of the possible latin table encoding, which are identified with their number (from 1 to 16, to the exception of 12).
All lexicon entries from the XIP lexicons, are automatically recorded in UTF8, the conversion is done from Latin to UTF8. Since, words can be encoded in different flavours of Latin, you need to provide in which Latin table the strings are encoded.
If the lexicons are already in UTF8, then nothing new will happen. The strings from the input will be also converted into UTF8 before comparison.

convert_lexicon_to_utf8_from_latin: 1 (table is Latin-1)

Rule Compilation Directives (#ifdef, #else,#endif)

XIP provides a system, which is very similar to the C language, to control which part of grammar might be compiled.
These directives are: **#ifdef**, the **#else** and the **#endif**.

These directives can be used in any file to encapsulate lines that should be compiled or skipped.
The *#ifdef* bears on values that should be declared in the GRM file in *variable* field. These values can be combined into Boolean expressions, with the following operators: &, |, ~.

Exemple:

GRM file:

```
variable:
TOTO //only one value should be declared at a time for each field
variable:
TITI
```

Grammar File

```
Directive:
/#ifdef (TITI | TOTO)\        //the directive should be enclosed with "/…\"
… code
/#else\
…code
/#endif\
```

More than one directive can be embedded into a file.

```
/#ifdef (TITI)\
…
/#else\
/#ifdef (TOTO)\
…
/#endif\
/#endif\
```

Rule Space Names

A user can declare rule space names into her/his grammar. These rule space names are then used at run time to activate or deactivate large section of the grammar.

Declaration

A Rule Space name is a string, which starts with a common root to all rule space names in the grammar. This common root is defined by the user. This rule space may then be refined with subsequent strings, separated with a ".". A rule space always starts with a #@*rulespace* instruction followed by a string, where names are separated with a ".".

Example:

#@**rulespace**=english
…rules…
#@**rulespace**=english.chunker
…rules…
#@**rulespace**=english.chunker.tagger
…rules…
#@**rulespace**=english.chunker.sequence

In this example, the root is "english" with four rule space derivations. "english" should always be used in all further rule space declarations.

Important

The same rule space name can be used at many different places in different files in a grammar, thus builds a logical view of the grammar.

Deactivate a rule space

A rule space can be deactivated with the "-block" instruction followed with a rule space name on the command line.

Example

xips …. –block english.chunker

When a rule space name is mentioned on the command line, then all derived rule spaces are impacted. For instance, if "english.chunker" is used, then the rule spaces english.chunker, english.chunker.tagger, and english.chunker.sequence are deactivated.

A rule space can also be deactivated or activated from the grammar itself with the script instructions *free* and *block.*

Example

Script:
block(“english.chunker.sequence”);

#@comment=…@

This field is dedicated to contextual menus. It records a comment on a given rule that is stored and associated to that rule. A comment should always end with a “@”. If this character “@” appears in the comment it should be preceded by a “\”.

Example

#@comment= This is a comment.@
Rules…

Function Name Declaration With Features

Function names can now be described with a specific set of features that is automatically instantiated to any new dependency that is created.

Below is an example of how this description should be made:

```
Functions:

subj = [fsubj=+].
obj.
varg.
determiner = [fdet=+].
```

N.B. The previous way of declaring dependencies is still active. Contrary to the categories (which are described in the same way), the system does not impose that a dependency should be declared together with specific features.

Creation of Phrasal Nodes Readings

Until now, XIP only accepted multiple readings at the word level. Now, a phrasal node can be associated with a list of possible readings. This is done through a specific operator in the "where" field. The previous affectation rule: #0[..] ={...} did associate with the mother node under construction, a computation of features out of a list of given nodes, for a certain type of features. The novelty here is the possibility for a new node to receive a set of common readings (and not features) from 1,2 or more nodes. The affectation rule can now be written as: #0<agreement> = {#1 & #2 &...}. This new rule computes the intersection of common readings between #1 and #2 on the basis of agreement, and creates as many readings as possible for the new mother node.

Example:

```
10> NP -> det#1[first], noun#2[last], where( #0<agr> = {#1 & #2}).
```

Comparison between Nodes on their readings.

In the same way, the comparison operator "::" between two nodes can also be used to compare whether two nodes have common readings.

Example:

```
where (#1<agr>::#2<agr>)
```

this above test checks whether #1 and #2 have readings that share the same values for the agreement features.
Dependency Rules

Layered Dependency Rules

The dependency rules can be declared in the chunking rule section. The only difference with other dependency rules is the presence of a layer number in the declaration of the rule.

```
1> if (…) D1,D2.
```

Instructions

Instructions with no Boolean value, such as print, _goto, calculus can be written directly in the body of the grammar. Each instruction should then end with a “;”.

Example:

```
If (Subj(#1,#2) & Obj(#1,#3)) ARG(#2,#1,#3).
print(“NEW rule\n”);
_id = _id +1 ;
```

List of instructions

If a sequence of instructions is to be triggered by a test or a TRE, then the sequence should be enclosed with “{}”. Each instruction should then be ended with a “;”.

Example

```
|Noun#1| {_num=_num+1; print(_num);}
```

IfTest keyword

The “IfTest” keyword is used when a list of instructions is needed according to a given condition. The instructions should be packed together between two “{}”. The “else” can be used in this clause. If a Boolean expression is present among the instructions, then the failure of that Boolean expression will not stop the process.

Important

A “iftest” can never be used to generate dependencies. This can only be done with a “if”.

NB

The “iftest” can be replaced with a “if”.

Example

```
Iftest (subj(#1,#2)) {
        print(“A subject is available”); _
        numbersubject = _numbersubject+1;
}
else {  print(“No subject available.”);}
```

Loading:

The loading field is a bit complex to grasp. This field is used to test some information while the rule file *is being loaded and compiled into memory.*

Example:

```
Loading:
If (@version~: "9.28") {
        error("Wrong version of the engine for that grammar\n");
}
//The error instruction will abort the loading of the grammar.
```

Initial:

The initial field is used to process certain instructions (such as storing XML files built on the fly) at the beginning of each new input that is being processed (such a new string or a new file).

Example:

```
Initial:
        i=0;
//We initialize a variable before any new parsing.
```

Final:

The final field is used to process certain instructions (such as storing XML files built on the fly) at the end of the processing of a full document.

Example:

```
Final:
        @_current(root)->Store("filename");
```

The file "filename" is stored at the end when the document has been fully processed.

Tail:

The *tail* field is used to force the execution of certain instructions at the end of the analysis of a sentence. The difference between *tail* and *final* is that *tail* does not depends on the value of *upto*, which defines up to which rule the grammar should apply to a piece of text. In the context of a debugging session, it can be quite useful to have these instructions being executed in all circumstances.

Dependency Variable Management

The scope of dependency variables ($xxx) has been enlarged to authorize some new behavior. It is possible to delete more than one dependency at a time, or to mix in the same rule the deletion and modification of dependencies.

Deletion

The deletion operator is still the "~" operator, which now can be followed with a dependency variable: ~$1.

Modification

The modification operator is the "^", which can be followed a variable or a new expression based on that variable.

More than one modification or deletion can occur at a time.

Example 1

```
if (subject$1(#1,#2) & object$2(#1,#3)) ~$1, ^subj$2(#1,#3).
```

The above rule reads: if a subject has been extracted together with an object for the same verb, then the subject is deleted (~$1) and the object is rewritten as a subject (^subj$2(#1,#3)). The link is made through the dependency variables $1 and $2.

Example 2

```
if (subject$1(#1,#2) & object$2(#1,#3)) ~$1, ^$2(#1,#2).
```

The above rule reads: if a subject has been extracted together with an object for the same verb, then the subject is deleted and the parameters of the object are replaced by those of the subject. In this case, the dependency name is not modified.

Example 3

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

One should be careful with those new operators. For instance, the above rule still deletes the subject dependency, but also creates a new object dependency with the same parameters as the previous subject. This rule is equivalent to:

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

Important

The previous caret "^" operator in the "if" side of the rule is still valid. It is possible to mix both approaches in the same time. It is important to remember that when using the "^" operator in the left side of the rule, then the first element in the result side of the rule corresponds to the action that is applied on the selected dependency. Be careful not confuse variables and selected dependencies when using both operators in the same time, as the result may not be coherent.

Category Definition Rules

The category definition rules are used in collaboration with the translation rules. Their goal is to solve the category definition of a specific entry for which no syntactic categories has been found after the application of the translation rules.

The keyword is *categorydefinition* or *définitioncatégorie.*

A category definition rule is described as below:

CategoryDefinition:

[features] = Category[new_features].

More than one rule can be described at a time. They applied in sequence, when a rule applies, then the system stop. The *new_features* are optional.
A category definition rule reads as follow:
The left-side of the rule is a constraint on the features extracted so far from the initial entry. If this constraint is verified then the category together with the features of the right-side of the rule are assigned to that entry.

Example:

```
CategoryDefinition:
        [gender] = Noun.
```

If the feature gender were found for an entry with no category, then this entry is a Noun.

How it works

If the FST lexicon yields the following output:

```
Word  word   +Noun+Sg+Masc
```

We first apply the translation rules on this entry to translate each substring starting with a ‘+’ as a feature or a category. If none of these substrings is recognized by the translation rule as a syntactic category, then the XIP representation of that entry might be underspecified.
We use the category definition rules, which apply the left part of these rules to detect a specific feature configuration. It this feature

configuration is detected, then the entry receives the category and the features described on the right-side of the rule.

Feature Definition Rules

The feature definition rules are used in collaboration with the translation rules. Their goal is to provide XIP with a very basic mechanism to enrich a feature structures with other features.

The keyword is *featuredefinition* or *définitiontraits.*

A feature definition rule is described as below:

FeatureDefinition:

[features] = [new_features].

More than one rule can be described at a time. They applied in sequence, when a rule applies, then the system stop.
The left-side of the rule is a constraint on the features extracted so far from the initial entry. If this constraint is verified then the features of the right-side of the rule are assigned to that entry.

Example:

```
FeatureDefinition:
        [fem:+,sg:+] = [fs=+].
```

If the features *fem* and *sg* are present on a reading then we also add *fs*.

Token Selection Rules

The token selection rules are used for languages in which a token can have different interpretations either as one token or as a sequence of tokens.

For instance, the Portuguese word: *pela* can be interpreted as:

a) Verb: *pelar*
b) Preposition followed with a determiner: *por o.*

However, the selection of which interpretation to choose is often difficult. The *Token Selection Rules* are used to select which interpretation should be kept. The rule matches the following format:

Tokenselection:

12> |left context| X1, X2,..Xn |right context|, where (…).

X1, X2,…,Xn are the categories which correspond to the expansion of the token. This sequence follows the sequence rule pattern.

Fragmentation Rules

The fragmentation rules are used to split a lemma form into a list of substring, which each receives a specific category/features described in the rule.
Fragmentation rules are introduced with the *fragment* keyword.

A fragmentation rule reads as follow:

```
Fragment:
        Rgx :  Cat[feat] = Cat%1[feat],..,Cat%n[feat].
```

1. Rgx is a XIP regular expression, where each substring is tagged with a specific %n variable.
2. Cat[feat] is a constraint on the category and the feature of the word this rule should apply on
3. For each %n variable, there should be a matching Cat%n.

After the application of that rule, the word is split into n substring each associated with a category and a set of features according to the variables associated to each element of the rule.

Example

?+*%1*lo*%2* : noun = noun%1,pron%2.

If we apply this rule on Noun[lemma:*cavalo*]:

1. *%1* is associated to *cava*.
2. *%2* is associated to *lo.*

The rule creates the two following syntactic nodes in the chunk tree:

Noun[lemma:cava], pron[lemma:lo]

Multiwords (*Multilemmas)* Rules

The purpose of these rules is to detect at the sentence level a sub-string that could be read as one token.
A multiword rule is introduced in a rule file with the keyword: **multiwords** (or **multilemmas)** (*multimots* or *multilemmes* in French). It is composed of a string or a regular expression followed by the "=" sign and the category which should top it.

Multiwords:
layer> "regular expression" = Cat[feat].

When XIP applies these rules, it builds a string out of the top nodes of a chunk tree. This string is then matched against the *multiword* rules. This string only comprises surface forms.
If the top nodes have already been replaced with categories, in other words, if chunking rules have already applied, then the categories of these nodes is used instead of the surface forms.
If *multilemmas* is used, the system will use the lemmas instead of the surface forms.

Example:

If the following sentence has been partially analyzed into the following chunk tree:

Det{The} Noun{lady} *VP{Verb{eats}}* Det{a} Noun{cake} Punt{.}

Then XIP first builds the following string:

The lady *VP* a cake .

The multiword rules are then applied on that string.

It should be noted that strings in a multiword layer are always applied before complex regular expressions:

10> "silent ?* movie" = Adj[multi:+].
10> "silent movie" = Noun[multi:+].

The second rule always applies before the previous one. To impose that regular expression rules apply before other rules, it suffices to move them to a previous layer.

DoubleReduction Rules

The double reduction rules allow for a parallel simplification of readings on two entries. Such a rule comprises two parts, one is applied on the sister categories on the top of the chunking tree, the second one applies a boolean expression on those nodes which is used as a pattern to select those readings that are compliant with that boolean test.

Important: the system applies that reduction to the nodes that are associated with the variables: #1 and #2. Other variables can be used within the boolean expression but their readings will not be reduced. Also, the regular expression pattern that is used to extract the nodes is not deterministic.

Example:

```
|det#1, ?*[verb:~,noun:~], noun#2,?#3[verb]| => #1[agr] :: #2[agr] &
#3[pers:3].
```

The first part of the rule extracts all nouns that are preceded by a determiner, while the second part of the rule only keeps those entries that share the same values for the agreement feature.

Term Rules

The term rules are a variation on the sequence rules. They should be declared in a layer. The keyword to declare those rules is: “Terms”.

A term rule is a sequence rule indexed on a lemma.

Declaration:

A term rule is composed of a lemma followed by “:” and then by the rule itself:

layer> Lemma : cat = cat0,…,catn.

If *lemma* is found in the sentence, then the system applies this rule. If the rule succeeds then a *cat* is built on the top of the *cat0…catn* nodes.

Important:

Cat0 always corresponds to the node associated with the lemma.

Backward Rules

A second sort of rule is available, the backward rule. A backward rule starts on the rightest category and try to apply the rule backward:

layer> Lemma : cat <= catn,…,cat0.

In this case, *lemma* is the last element of the rule.

Example:

Terms:

2> as : prep = prep[start], adj[lemma:long],prep[lemma:as].

This rule recognizes the expression “as long as” and transformed it into a preposition, only if it starts the sentence.

Semantic rules

Semantic rules are dependency rules that are triggered by the presence of a lemma in a sentence. By definition, the lemma that has been detected is always *associated to the variable #0.* The keyword is “semantic”.

Declaration

Lemma : Cat[features] = if (…) D1,..,Dn.

If *lemma* has been found in a sentence with *Cat[features]* as a category and features, then we apply the dependency rule associated. These rules are indexed in memory in an efficient way and a large number of them can be managed at once. If the number is to large, then XIP may be very slow at loading time, it could be useful then to use the indexing mechanism which is described in the next section.
More than one rule can be associated to the same lemma. They will all apply.
The “if” part in the rule is optional.

Example

a) In the rule below, if we find the noun *abolitionist*, we create the dependency *addition* with “abolitionism” in the stack.
abolitionist: Noun = *addition*[$STACK="abolitionism", adj=+](#0).

b) In the rule below, if we find the word *absent* in a sentence in a *noun modifier* configuration (NMOD), we then build the dependency *VARG* (*verb argument)* with the verb “absent”.

absent : Noun = *if*(NMOD[INDIR](#0,#1,#2))
VARG[DIR](##Verb[lemma=absent],#2).

Indexing

XIP allows now an indexing of semantic disambiguation rules through the generation of an index file. This index file comprises an ordered list of all the words that those SDR are indexed on. Each of these index words are then associated with a global position whithin the file that stores those rules. After the generation of that index file, the rule files should not be modified.

Creation

To create such an index file, the command is the following:

```
xip -sem Rule1.xip -sem Rule2.xip -index File.index
```

File.index is used to store the index that is computed out of the semantic files.

Important: the pathname of the semantic rule files should be absolute, since those pathnames are stored in the index file.

Example:

```
xip -sem /home/user/grammar/semantic1.xip -sem
/home/user/grammar/semantic2.xip -index semantic.index
```

N.B. XIP manages the semantic files into specific layers. If one computes an index for several rule files, then those files are merged together in a common index. If one wants to keep those files in different layers, so that the rule of the first file applies before the rule of the second file, the solution consists of generating a different index for each file and to load those indexes according to their precedence.

Loading

The index is loaded by with the -p command.

```
xip -p File.index
```

The other files which this index is based on should not be loaded in memory. More than one index may be managed at a time by XIP.

```
xip -p file1.index -p file2.index
```

Each of those indexes puts in a different layer the rules that have been indexed. So the rules indexed by file1.index apply before the rules indexed by file2.index

Creation of a virtual node

It is now possible to create a dependency whose parameters are not all based on the chunk tree. In other words, a dependency can create some of its parameters as a sort of virtual node.

The creation of such a node is commanded by the following rule: ##Cat[features].

The command is ## followed by a syntactic category associated with some features.

Example:

|NP#1, FV#2| subject(#2,#1,##Noun[lemma=myword]).

This rule creates a subject dependency with three parameters, the last being a new node that is created on the fly.

The result is:

John sells a horse.

subject(sell, John, myword).

This node does not appear in the chunk tree, but behaves exactly as a syntactic node. However, it is transmitted through the API and in XML as node with the left offset=-100 and right offset==-10.

Stack Management

Dependencies (and only dependencies) can be associated with a specific variable: $STACK. This variable is used whithin the feature structure associated to the dependency and stores in a stack style any sort of strings.

The $STACK can be used with the following operators:

$STACK= value	we push on the top of the stack a given value
$STACK : value	we verify if $STACK comprises a given "value" among all its elements.
$STACK:~value	We test whether $STACK does not store any "value"
$STACK=~value	We remove "value" from the stack
$STACK=-num	We pop $STACK of "num" element
$STACK=~	We clear $STACK
$STACK:~	We check whether the $STACK is empty.

Example:

```
//We push two strings on the stack
dependency[$STACK=essai, $STACK=truc]

//We test whether one of the values pushed on the stack is essai
dependency[$STACK:essai]

//We remove one value on the top of the STACK
dependency[$STACK=-1]

//We clear the stack
dependency[$STACK=~]
```

Permanent Features

XIP provides also a way to make certain features permanent. These permanent features cannot be deleted nor modified once they have been defined. Certain XIP features are permanent per se.

Permanent XIP Features

The following features are permanent per se:

a) *start, end*, which denotes the first and the last word of a sentence
b) *uppercase, alluppercase*, which are automatically instantiated on word that starts with an uppercase or only composed of uppercase characters.

Markup Tag Features

The markup features, which are either defines in the XIP lexicons, or through a XML *feature* instruction are automatically declared as permanent.

Permanent Feature Declaration

In the other cases, a feature value may become permanent in a lexicon entree, a translation rule or any other rules within a feature structure thanks to the ":=" operator.

Example:

time += [ftime:=+]. *//ftime is then a permanent feature*

Variables

XIP provides now a new way of dealing with numbers. One can now define strings, arrays, numerical variables, integer or float, attached to a node or a dependency. Those variables can be modified with numerical expressions. A limited set of operator is available at present but can be easily modified and enlarged.

Variables

The new field name is "Variables" (in English and in French).

Types

The type of the variable is:

"int" for integer
"float" for real.
“string” for string.
“vint” for a vector of numbers (equivalent to: int _x[])
“vector” for a vector of string (equivalent to: string _x[])
“vindex” for a vector of string associated with an internal index.
“dictionary” for a vector which is indexed on strings
“struct” which is used to create new groups of variables
“python” which is used to keep track of a python variable throughout a grammar
“xml” which is used to handle xml nodes.

N.B. There is no real difference in the use of vindex v.s. vector. vindex is slower to feed as an index is automatically built for each internal string, but is much faster to search.

IMPORTANT

In the case of a function call, where two integers would be separated by a comma as in: *insertvalue(vect,1,2);* The integers should be separated with a “;” instead of a *“,”.*

The correct expression is: *insertvalue(vect,1;2);*

Different sorts of variable

There are five sorts of variables.

Global Variables

Three sorts of global variables: one that is reinitialized for each new sentence that is analyzed, the other sorts that keep their value from the start to the end of the analysis. The latter is indexed on the lemma string of a node.

a) The global variables that are never reinitialized should always start with a "_".
b) The global variables that are reinitialized for each new sentence are a simple string.
c) The node string variables (these that are indexed on a lemma string) should always end with two "##".

Node Variable

A node variable is attached to a specific node in the tree.

The node variables always end with a "#".

Dependency Variable

A dependency variable is attached to a specific dependency
The dependency variables always end with a "$".

Declaration

A variable declaration should always ends with a ";". The initialization of the variable can be provided with an "=" sign. The formula that follows the "=" sign is used to initialize the variable according to its type. For instance, this formula is used just once for the global variables and for each new analysis for all the others.

N.B. The vint and the vector can also be declared in the following way:

```
vector _x; is equivalent to:          string _x[];
vint _y; is equivalent to:  int _y[];
```

Struct name {…}

The "struct" is a structure à la C++, which can be used to create complex variables.

Example:

We create a “struct” that comprises a string and an integer:

```
Struct mystruct {
        int val;
        string str;
}

mystruct _foo[]; //we create a vector of mystruct object
mystruct _x;     //we create one instance of mystruct
```

Script:

```
//We use it in the following way:
_x.val=0;
_x.str=”toto”;

_foo[0].val=0 ;
_foo[0].str= “titi”;

for (i=0;i<_foo;i++) {
        print(_foo[0].val);
} //_foo contains the size of the vector

_foo=0; //All the elements of the vector are deleted
```

How to use these variables?

Those variables can handled in "where",”iftest” or "if" expressions.

N.B. If a deduction rule is reduced to a simple manipulation of those variables, the "compute" keyword can be used instead the "if".

The different operators

XIP provides the following operators. (This list can be modified and enlarged if necessary).

The basic operators: +,-, *, /

Some more refined operators:

1. x^y (x to the power of y)
2. x%y (x modulo y)

Some numerical operators:

1. **ln**(x) the natural logarithm of x
2. **log**(x) the logarithm in base 10 of x
3. **exp**(x) the exponential of x
4. **sqrt**(x) the square root of x
5. **tan**(x) the tangent of x
6. **sin**(x) the sine of x
7. **cos**(x) the cosine of x
8. **atan**(x) the arc tangent of x

The Lemma Variable Functions

The lemma variables (which are indexed on a lemma) are very powerful variables which can accumulate in memory. Being indexed on a lemma, they are also quite difficult to access without any specific lemma available. We provide different methods to palliate this problem.

1. **cleanlemmas**(var##) This function is used to delete the lemma variables from their possible instantiation. Each time a lemma variable is used for a given lemma, a new variable is created in memory, which is never deleted. This can end up into an over-accumulation of variable, which are never removed from memory unless one uses this operator. This is especially useful if more than one document is analysed at a time.

2. **cleanAllLemmas**: This function deletes the lemmas for all lemma variables that have been declared in the grammar.

3. **cleanAll:** This function reset all variables in memory except the lemma variables.

Example

int lem##;

Script:
|Noun#1| {lem##1=1;} //we create a new variable for each noun

Final:
Cleanlemmas(lem##); //at the end of the analysis of the document, these variables are deleted from memory.

3. R**etrieveLemmas**(var##,_vstr,_vt) This function extracts for a given lemma variable, all the specific variables built on the top of it, together with their value. "_vstr" contains the name of all those variables and "_vt" their value. "_vstr" should be a "vector" variable, and "_vt" a vint

variable. This function returns the number of variables actually extracted.

4. **GetLemma**(var##,str) This function returns the value of a lemma variable for a given lemma string.

Some String operators:

1. **len**(string) the length of the string
2. **left**(string,nb) returns the first nb characters on the left of string
3. **right**(string,nb) returns the last nb characters on the right of string
4. **mid**(string,pos,nb) returns the nb characters starting at position pos
5. **lowercase**(string) return the string in lowercase
6. **uppercase**(string) return the string in uppercase
7. **removefirst**(string,n) removes the first n characters from string
8. **removelast**(string,n) removes the last n characters from string
9. **trim**(string) removes the trailing characters (whites spaces)

System Functions

1. **goto(str)** Goto label str
2. **exit** **S**top the current analysis
3. **return(x)** Return a numerical value x from a procedure
4. **Stop** See exit
5. **error(str)** Stop the XIP engine and generates an exception with the error message str
6. **getenv(var,env)** Get the string value of the environment variable env into var
7. **putenv(env,var)** Set the string value of the environment variable env from var
8. **@lasterror** **R**eturn the error number of the last error
9. **errormessage(i)** Return the error string corresponding to i
10. **eval(s,layer)** **E**valuate *str* as a grammar file. *layer* is a int: 0 means absolute layering, 1 means relative layering.

Regular Expressions

1. **regexp**(string,rgx) returns the substring corresponding to the regular expression *rgx*. If the variable that receives the result is a “int” or a “float” then regexp yields the position of the first

character starting the sequence. This regular expression can match anywhere in the string.

2. **regexplast**(string,rgx) Similar to **regexp**. It returns the first substring corresponding to the regular expression *rgx* but starting from the end of the string.
3. **regexpstart**(string,rgx) returns the substring corresponding to the regular expression starting at the beginning of string.
4. **regexpback**(string,rgx) returns the substring corresponding to the regular expression back from the end of string.
5. **regexpfull**(string,rgx) returns 1, if the regular expression applies to the whole string. 0 otherwise.
6. **regexpsplit**(string,rgx,_vect) stores in the "vector" variable _*vect* all the occurrences of the regular expression "rgx" over the string.
7. **replace**(string,"rgx","nstr") replaces each substring in "string" matching the regular expression "rgx" by the string "nstr". This instruction returns the modified string.
8. **splitalong**(str,cut,vect) cuts the string *str* along the string *cut* and stored the different values in *vect*. In no substring *cut* is present in *str,* then *vect* will contain only one value, *str* itself.

Accessing characters,values and strings

1. **search**(_vect,string) returns the position of string in the vector, if this string is already present in the vector. Returns -1 if no string is found.
2. **search**(_vect,string,_res) stores in _res the list of all indexes of all occurrences of string in _vect. Returns the number of string found. _res must be a vector or a vindex.
3. **searchvalue**(_vect,int) search for a numerical value, _vect must be a vint.
4. **searchvalue**(_vect,int,_res)
5. **_str[i]** "_str" must be a string. Returns the character in position "i" in the string "_str".
6. **_vect[i]** "_vect" must be a vector. Return the string in position "i" in the vector "_vect";

Handling dictionary

A "dictionary" is a specific sort of vector where the access is done with a string.

Example:

```
Dictionary dico;

dico["first"]="one";
dico["second"]="two";
```

None of the other functions, which apply on vectors, can be used with this type, to the exception of:

backsearch(dico,"string",vect) This instruction retrieves the list of indexes that are attached to the "string" value.

Keys(dico,vect) this instruction returns the vectors of keys (the value between square brackets)

Values(dico,vect) This instruction returns the values of the dictionary.

KeyOnIndex(dico,i) This instruction returns the key string corresponding to the ith key value in dico.

It is also possible to loop in a dictionary with an integer value:

Example:

```
For (i=0;i<dico;i++) {
        key=KeyOnIndex(dico,i);
        print("Element:"+dico[i]+":"+key+"\n");
}
```

KeyOnIndex can be used to retrieve the key corresponding to a specific digital index.

Handling vector or vindex

1. **insert**(_vect,"string",pos) inserts the string "string" in _vect at position "pos". The other strings are shifted to the right.
2. **addstring**(_vect,string) add a string at the end of the vector. Yield the index of the string in the vector.
3. **addstringunique**(_vect**,**string) add a string at the end of the vector. If this string is already stored in _vect, yield the current index of this string. This instruction is used to avoid duplicating entries in the vector.
4. **removestring**(_vect,i) remove the "ith" element in the vector.
5. **addvalue**(_vect,int) add a value to a vint vector
6. **addvalueunique**(_vect,int) add a value to a vint vector, only if this value is not already recorded.
7. **insertvalue(**vect,val,i) inserts the value "val" at the position "i" in vect.
8. **removevalue**(_vect,i) remove the "ith" element in a vector.

File Management

It is also possible to open a text file to write string in it. XIP provides three primitive to open, read and close a text file.

1. **fopen**(alias,pathname) Open the filename “pathname” and associate it to the *alias*. *Alias* and *pathname* should be provided as strings.
2. **fwrite(**alias,string) Write the string *string* to the file corresponding to alias
3. **fclose**(alias) Close the file associated to *alias*

Time

It is possible to get some information about the date:

1.	**@date**	returns the date as a string
2.	**@monthstring**	returns the month as a string
3.	**@daystring**	returns the day as a string
4.	**@hour**	returns the hour as a string
5.	**@year**	returns the year number
6.	**@day**	returns the day number in the month
7.	**@month**	returns the month number

It is possible to measure the elapsed time for a given script.

1. **starttimer** Starts the timer.
2. **endtimer** Ends the timer.
3. **@elapsedtime** Returns the elapsed time
4. **@intermediate** Returns the intermediate elapsed time

XML manipulation

The XML nodes can be manipulated with specific instructions, which are listed below.

This type is implemented as a placeholder for the *xmlNodePtr* type from the *libxml2 library* (see http://xmlsoft.org/), hence the *new* method which is necessary to get a new object for the current variable.

1. xmlattributes(#1,vdict) Returns the attributes associated to the XML token, which was used to create that lexical node.
2. **xmldata**(vxml,vdict) Returns the data associated to a XML node variable into a dictionary
3. **xmlparent**(vxml,vxmlparent) Stores in vxmlparent, the parent of vxml
4. **xmlchild**(vxml,vxmlchild) Stores in vxmlchild, the vxml child
5. **xmlnext**(vxml,vxmlnext) Stores in vxmlnext, the next node of vxml
6. **xmlprevious**(vxml,vxmlprevious) Stores in vxmlprevious, the previous node of vxml
7. **xmltext**(vxml,str) Stores in str the text of the XML node vxml
8. **xmltag**(vxml,str): Stores in str the tag name of the XML node vxml
9. **xmlset**("@reference",vxml): Stores back in @reference, the value of vxml
10. **xmlstring(string tag,string c):** Return a string containing an XML node whose name is tag with c as content.

Rule Space Instructions

1. **block**(str) Deactivates the str rule space. Returns 1, if it succeeds.
2. **free**(str) Reactivate the str rule space. Returns 1, if it succeeds.
3. **freefull**(str) Reactivate the str rule space. Returns 1, if it succeeds. This function also reactivates the upper nodes in the hierarchy.
4. **blocked**(vect) Returns the vector of deactivated rule space strings. Returns the vector size.
5. **rulespace**(vect) Returns the vector of all declared rule space string. Returns the vector size.

Example

//We declare some xml variables:
Variables:

```
        xml current;
        xml nextxml;
        string xmlname;
```

Script:

```
//The following instruction is used to recover the xml node pointer of
the //current node
@_current()->set(current);

//We then take the parent node, if it possible
If (xmlnext(current,nextxml):1) {
        xmltag(nextxml,xmlname);
        print("Tag="+xmlname+"\n");
}
```

Error Handling

last_error() yields the message number. Returns -1 if no error was detected.
message_error(id) returns the error message corresponding to last_error).

Some important points about a vector or a vindex

A vector should be used with "[id]" with id being the index of the element. The variable used without any "[]" returns the size of the vector.

Operations on nodes

Offset(#1,left,right) returns the left offset and the right offset of the node #1
Offsetchar(#1,left,right) returns the left offset and the right offset of the node #1 in characters.
Tokennumber(#1,left,right) returns the left and right token numbers of the node #1
i=**Nodenumber(**#1) returns the node number of #1 into i.
Node(#1,n) assign to the variable #1 the node identified by the node number n.
Copyfeature(#1[feat],#2) copies the features "feat" from #1 to #2, also works with dependency variable,
SetFeature(#1,attribute,value) set the *attribute:value* feature to #1. *attribute* and *value* should be strings.

SetSubFeature(#1,attribute,value) set the *attribute:value* feature to #1, and to all its sub-nodes. *attribute* and *value* should be strings.
Lemmas(_vect,#1) stores in "_vect" all the possible lemma readings for #1.
i=**Dependencynumber**($1) returns the index of the dependency $1
Dependency($1,i) assigns the dependency whose index is *i* to the $1 variable.

push($1,str)	pushes the string str on the top of $1's stack
pop($1)	pops the element on the top of $1's stack
cleanstack($1)	clears $1's stack

createdependency($1,name,features,#1,#2,…) Create a new dependency with *name* as a string, *features* as a string (between brackets: "[feat=val]"), and as many nodes as necessary. The dependency is then stored in the $1 variable. $1 and features are optional.

Some basic values:

@P	PI value
@E	The base of natural logarithms e.
@pathname	The pathname of the file that is currently parsed
@sentencenumber	the current sentence number
@elpasedtime	returns the time that elapsed between a starttimer and a endtimer.
@xip	returns the XIP engine version
@version	returns the grammar version
@language	returns the language that was detected for a given sentence.
@wordcount	returns the number of words in the current sentence
@punctcount	returns the number of punctuations in the current sentence

Some lemma and surface manipulation

It is possible to use some string expressions to modify or to test the lemma or the surface form of a given node.

Example:

```
|Noun#1| {
        _str=left(#1[lemma],len(#1[lemma])-3);
        #1[lemma=_str];
}
```

Important:

It is possible to also use string functions in that context, but these string functions cannot make any reference to a node variable.

Example:

```
|det#1,Noun#2| {
        #2[lemma=left(#1[lemma],2)];
} //This is invalid…

|det#1,Noun#2| {
        _str= left(#1[lemma],2);
        #2[lemma=_str];
} //This is valid…

|det#1,Noun#2| {
        _str= #1[lemma];
        #2[lemma=left(_str,2)];
} //This is also valid…
```

How to display the results?

The results are not automatically displayed on screen. One should use the "-math" option on the command line to see the results for those variables. However, if XIP is used in the XML mode or with its API, the variables are automatically incorporated as features associated to their nodes or to their dependencies. The global variables are always attached to the TOP node.

Examples

Declaration

```
Variables:
//The variable below is a global variable that is never initialised. Its default initialisation value is 0.
int _globalneverinitialised;

//The variable below is a global variable that will initialised for each new anaylsis. Its initialisation value is -1.
int thisaglobal=-1;
```

//The variable below is a node variable. This variable is always attached to a node. Its initialisation value is pi.
float node# = PI@;

//The variable below is a node variable. This variable is always attached to a lemma string, which is borne by a node.
float string## = 0;

//The variable below is a dependency variable. This variable is always attached to a dependency.

float depend$ = 10;

Example of a rule

Sequence:

NP = Det,Noun#1, where node#1 = node#1 + 1).

The above rule computes the number of Noun nodes for which an NP has been built. The variable "node#1" is attached to the #1 node through its number.
IMPORTANT: There would be as many "node#" variables as Noun nodes in the sentence. This mechanism can be used as a rough equivalent to a table.

Example of lemma variables

NP = Det, Noun#1, where string##1 = string##1 +1).

The main difference between the above rule and the previous one is that this variable is not indexed on the node but on the lemma that is found on that node. For instance, if #1 was attached to a node whose lemma is "dog", then this variable would be attached the string node. Each time, a node with the lemma "dog" is found, this variable is incremented by 1. In this case, the above rule counts the different occurences of the nouns that are spanned by this rule.

Example of a string variable

string _var;

_var = #1[lemma]; //We store the lemma in _var

|noun[lemma:_var]| //we compare a lemma with _var.

|noun#1| NewDependency(##Noun[lemma=_var]). //We create a new fictive noun with a lemma whose value is stored in _var.

Example of a vector variable

```
vector _var;
int _id;
_id=0;
#insert
iftest (_id<3) {
    _var[_id]=”chaine”+_id+”\n”;
    _id=_id+1;
    _goto(“insert”);
}

_id=0;
print(“We display:\n”);
#display
iftest (_id<_var) {     //the variable _var returns the size of the vector
    print(_var[_id]);
    _id=_id+1;
}

//The result is:
We display:
chaine0
chaine1
chaine2
```

LOOPS

XIP also provides loop mechanisms.

FOR(i=0;i<10;i=i+1) {…}

The “for” is used to automatically increment a numerical variable.

Example

```
for (i=0;i<=100;i++) {
        printi);
}
```

While(condition) {…};

While the condition is true, the internal instructions are processed.

Example

```
i=2;
while (i>2 & i<10) {
	print(i);
	i=i+1;
}
```

Whilexml(xmlcondition) {…}

This instruction is very specific to the treatment of XML instructions. If one want to loop among XML nodes, then this instruction should be used. The other instructions *iftest* or *while*) needs only to detect whether a certain condition on an XML node is true for at least one node. *Whilexml* loops among these nodes in order to gather all possible information about them. It should be used with @_currentxmlnode which always points on the last XML node detected.

Example

```
Whilexml((@myaxml(/Root/Test)->Test()) {
	@_currentxmlnode()->Assign(_str);
	Print("Content:"+_str+"\n");
}
```

if more than one /Root/Test node is available, then we will display the content of each of these XML nodes.

Lexicon Regular Expressions

XIP now provides a new way of declaring words in lexicons with regular expressions based on the lemma string of the word. The regular expression used in XIP can only be used to declare words in the lexicons.

Regular Expression Formalism

The meta-characters

A regular expression is a string where meta-characters can be used to introduce a certain freedom in the description of the word. These meta-characters are the following:

%d	stands for any digit
%p	stands for any punctuation belonging to the following set: **< > { } [ ]) , ; : . & | ! / \ = ~ # @ ^ ? + - * $ % ' _ ¬ £ €` "**
%c	stands for any lower case letter
%C	stands for any upper case letter
?	Stands for any character
%?	Stands for the character '?' iself

Example:

dog%c	matches *dogs* or *dogg*
m%d	matches m0, m1,…,m9

The operators *,+, () , ([])

A regular expression can use the Kleene-star convention to define characters that occurs more than once.

x*:	the character can be repeated 0 or n times
x+:	the character must be present at least once
(x):	the character is optional
([x,…,x]*,+):	defines a character that can have more than one property

where x is a character or a meta-character. There is one special case with the '*' and the '+'. If the character that is to be repeated can be any character, then one should use **"%+"** or **"%*"** .

Important

These two rules are also equivalent to “?*” or “?+”.

Example:

1) **a*ed**	matches aed, aaed, aaaed etc. the *a* can be present
0 or	n times)
2) a%*ed	matches aed, aued, auaed, aubased etc. any
characters	can occur between *a* and *ed*)
3) a%d*	matches a, a1, a23, a45, a765735 etc.
4) a[%d,%p]	matches a1, a/, a etc.
5) a[bef]	matches ab, ae or af.
6) a[%d,bef]	matches a1, ab, ae, af, a0, a9 etc.
7) a[be]+	matches ab, ae, abb, abe, abbbe, aeeeb etc.

GOTO, EXIT and Label

It is now possible to use gotos and labels in XIP.

Label

First of all, each rule can be associated with a “label”, which can be used to identify this rule in the trace file or in the “brackets” of a dependency when the “-ne” option has been selected on the command line.

Declaration

To declare a “label” , simple write before the rule itself a string preceded with a “#”:

```
#label
2> rule.
```

A label can be used declared for any sorts of rules, to the exception of the semantic rules and the lexicon declaration. A label is not case-sensitive: #LABEL is equivalent to #label.

GOTO(goto)

To use the “goto” instruction, one must first declare a label associated to a rule. The parameter of the “goto” is a string or variable string. It is possible to jump on computed labels.

IMPORTANT

You cannot jump from the chunking rule set to the dependency rule set. A jump is always either local to the layers, or local to the dependency rules declared with no layer number.

a) If the deduction rule is declared in a layer, you can only jump to rules declared with a layer number

b) If the deduction rule is not declared in a layer, then the “goto” can only branch rules with no layer number.

Branching Error

A branching error does not stop the grammar; a warning message is added to the trace file.

Example :

```
|Det,Noun| { goto(“verb”); }
If (test(subj#1,#2)) {goto(“dir”);}
…

#verb
…


#dir
…
```

exit;

This instruction stops the execution of the grammar for a given sentence. This instruction corresponds to “goto” end of grammar, which is always represented by a “$”. (Thus, exit corresponds to goto“$”).

Display(display)

The “display” command allows for a modification of the display mode on the fly within grammar rule. This instruction can also be used to modify the execution mode of XIP.

Display Value

Below is the list of all possible value to modify the display on the fly :

lemma	displays the lemma forms in the output
surface	displays the surface forms in the output
markup	displays the markup tags while parsing an XML file
category	displays the categories with each node
reduced	same as –tr on the command line
full	same as –a on the command line
offset	displays the word offset
wordnumber	displays the word numbers
sentence	displays the sentence
none	no display
standard	reset to the display mode to the original
depbyname	order dependencies by name
depbynode	order dependencies on the first parameter.
depbycreation	order dependencies by order of creation
sentencenumber	display the sentence number
rulenumber	display the rule number
math	display the mathematical variables
error	display the script execution error
screenxml	display the result of an XML file processing on screen
tree	display the chunk tree on screen
nop	Do not display any standard XIP results
chunk	Only chunks
dependency	Compute also the dependencies
tokenize	The input is a list of tokens
languagetest	Test the language for each sentence
utf8input	The input is in UTF8
utf8output	The output is in UTF8

Execution

Below is the list of all possible execution control values

nop	**No analysis**
chunk	**Only chunking**
dependency	**Also dependencies**

Combining

All the previous values can be combined in a call: display(full,chunk)

Example :

```
DependencyRules :

|noun#1,noun#2| {
        Display(nop);
        print("Pas d'execution\n")
}
```

Print(print)

The “print” instruction is used to display information on screen at parsing time. “print” can display strings, numerical values, features, dependency names, lemmas, surface forms, noun categories. It is possible to “print” specific messages in dependency rules to keep track of what is going on.

How to use it?

print can be used in six different ways:

a) It can print a string: print(“This is a string”)
b) It can print the node category: print(#1)
c) It can print a dependency name: print($1)
d) It can print the features of a node: print(#1[features])
e) It can print the features of a dependency: print($1[features])
f) It can print number variables: print(num)

Specific Characters: \n,\t,\r

\n, \t, \r are specific characters that can be used to add respectively: a carriage return, a tabulation and a line feed. They can be added in a string anywhere.

Important

You can mangle all these possibilities in one call, with the use of the “+”, which is the string concatenator.

Example

```
|Noun#1|
if ($1(#1,#2) &
    Print(“The dependency: “+$1+“\nThe features of that dependency:”+$1[features]) )
dep(#1,#2).
```

Procedures

XIP provides a way of declaring long sequence of tests as a procedure that can be called anywhere in a dependency rule.
The keyword is “procedures”.
Important: A procedure should always finish with a “return” statement.

Declaration

A procedure is declared as follow:

```
Procname(#1,..,#n, int, string) {
        …
        return(formula);
}
```

Variable management

If a variable in the call of the procedure is not bound at call time it is not associated with a node), then the procedure can set a value to that variable can associate it with a node).
One can also use string or integer variables in the call of a procedure.

Different sorts of Variables

There is a distinction in XIP, as in any computational language, between reference variables variables whose value is actually modified by the procedure) and value variables variables whose value is only passed to the procedure without any modification). The distinction between these two sorts of variables is done through a “_” at the beginning of the variable name. Any variable starting with a “_” is a reference variable.

Example:

```
Proc(inti, int j);  //i and j are local variables

Proc(int i, int _j) //_j is a global variable
```

Local Variables

One can declare more variables in the procedure than will be actually used in the calling of that procedure. These extra-variables are local variables.

Example:

```
Proc(int i, int j, int k)  {
        k=i+j;
        Return(k);
}

proc(4,5); //k is a local variable
```

Recursivity

A procedure can be recursively called; the user must provide a reasonable stop test to avoid losing the result of that call.

Predifined Procedures: mother, next, previous, daughter

XIP provides four pre-defined procedures to manage nodes. These four pre-defined procedures take as input a node variable and yield as result another node variable.

Important

If the result variable is already bound associated with a node), then these procedures are used as a way to check whether a node is linked within a specific configuration.

Mother(#1,#2)

This procedure takes as input a node #1 and yields as result the mother node #2 of that node #1. If #2 is already bound, then this method is used to check whether #2 is the mother node of #1.

Daughter(#1,#2)

This procedure takes as input a node #1 and yields as result the first daughter node #2 on the left. If #2 is already bound, the method checks whether #2 is the first daughter node on the left of #1.

Next(#1,#2)

This procedure takes as input a node #1 and yields the next node #2 of that node #1. If #2 is already bound, the method checks whether #2 is the next node of #1.

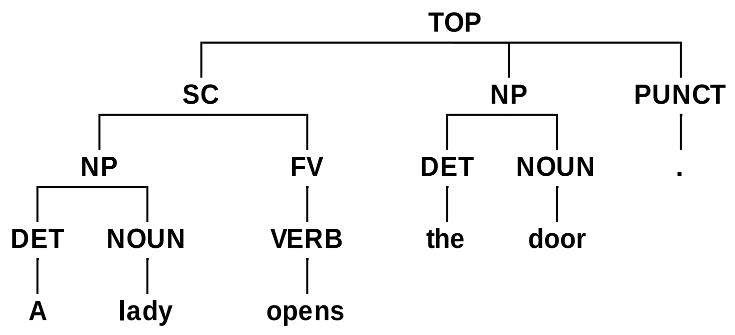
Previous(#1,#2)

This procedure takes as input a node #1 and yields the previous node #2 of that node #1. If #2 is already bound, the method checks whether #2 is the previous node of #1.

Descendant(#1,#2)

This procedure takes as input two nodes #1 and #2 and checks whether #2 is a descendant of #1.

Example 1



We suppose que #1 points on the node *NP{the door}*, and #3 points on *door.*
Then we have the following result:

Next(#1,#2)	#2 points on Punct
Daughter(#1,#2)	#2 points on Det{the}
Previous(#1,#2)	#2 points on SC
Mother(#1,#2)	#2 points on TOP.
Descendant(#1,#3)	returns TRUE

Example 2

We defined a procedure that checks if mother node of a given node is feminine.

Procedures:

```
Testmother(#1) {
        Iftest (mother(#1,#2) & #2[gender:fem]) {
                Return(1);
        }
        Return(0);
}
```

DependencyRule:

```
If (Testmother(#1) & subj(#2,#1)) dep(#1,#2).
```

We then use it in a dependency rule as a test.

Example 3

We define now a procedure that recursively tests whether all the mother nodes a feminine.

Procedures:

```
Testmother(#1) {
        if (test(mother(#1,#2)) {
                if (#2[gender:fem]) {
                        Return(Testmother(#2));
                }
                else {
                        return(0);
                }
        }
        return(1);
}
```

XIP XML

This part of XIP is implemented on top of the *libxml2 library* (see http://xmlsoft.org/).

The next sections are devoted to the XIP features about XML management. XIP allows one to parse an XML document using a specific grammar to drive that analysis. XIP allows one to use an XML file as a database. XIP can also be used to annotate a document with user-defined mark up tags. Finally, XIP can generate XML files on the fly to create specific repositories to store extracted information.

Important

I) Most of these instructions are based on XPath, which will not be presented in this document.

II) The XIP XML instructions can only be used with the *enabled XML* version of XIP. Please see with XEROX to verify whether your XIP version is an XML version.

Parsing an XML file: XMLguide

XIP offers a variety of tools to parse an XML file according to a specific grammar. This grammar is called an XML guide. An XML guide grammar defines for each markup tag of an XML document, how the content of that markup tag is to be analyzed. There are four possibilities:

a) The content is skipped
b) The content is tokenized
c) The content is chunked
d) The content is fully analyzed

If a markup tag is not defined in the grammar, then its content is analyzed according to the default behavior set by the user.

XPath

A XML guide instruction is always associated to an XPath. We will not describe the XPath syntax in this document.

XMLGuide: section name

The keyword of the XML guide section is XMLGuide.

XMLGuide instructions

Below is the definition of the grammar formalism that is to be used to parse an XML file. An XMLGuide grammar is stored in a text file that can be appended to the regular grammar. It is triggered by the use of "–xmltext" on the command line see in the command section is this document to see how to use it.)

Formalism

An xml guide instruction follows the rule below:

XPath|*#default*->skip(text|all).

XPath|*#default->feature(firstfeature,feature,token*|*dependency*|*chunk)).*

XPath|*#default->analyze(token*|*dependency*|*chunk).*

XPath|*#default->merge(token*|*dependency*|*chunk).*

XPath

A XMLguide instruction always starts with an XPath. If the markup tag is unique, then the “//” are optional.

Example

//Title->skip(text).
Title->skip(text).

The two instructions below are equivalent.

Instructions

XIP provides the following instructions to deal with markup tags content:

Skip(text|all)

Skip is used to tell XIP to skip the analysis of the markup tag content. *Skip* takes two possible values:

a) text: the content of the markup tag is skipped, but the sub-nodes are analyzed.
b) all: the content of the markup tag is skipped. The sub-nodes are not analyzed neither.

Example:

TextID->skip(text). *//the TextId content is not analyzed*

Feature(firstfeature,feature_marker,token|chunk|dependency))

This instruction translates a markup tag into a feature marking. All the words from the XML node content are marked with the feature *feature_marker.* Furthermore, the first word of the XML node content receives the feature *firstfeature.* Once this marking done, the content can be analyzed according to the third argument. This third argument is optional.

Important

Firstfeature and *feature_marker* must be declared in XIP as:
Firstfeature: {+},
Feature_marker: {+}.

Example

Bold->feature(boldfirst,bold).

DocId->feature(idfirst,id,token).

- **Analyze(token|chunk|dependency,"empty")**

The markup tag content is tokenized, chunked or full analyzed. The "empty" field is used to provide a string that is to be used if the *XML text* field is empty. This will ensure an execution of the grammar even if no text is available. This last field is optional.

Example:

Title->analyze(chunk). //*The title content is chunked.*

- **Merge(token|chunk|dependency,"empty")**

XIP merges all the sub-node contents together with the content of the main node in one single text that is analyzed at once. If one of the sub-nodes could match one of the XMLguide rules, then the merge declaration over-rules that rule and the content of that sub-node is analyzed according to the merge definition. The only exception is if this sub-node matches a feature rule.
The "empty" field is used to provide a string that is to be used if the *XML text* field is empty. This will ensure an execution of the grammar even if no text is available. This last field is optional.

- **Tokenize(token|chunk|dependency,"empty")**

This instruction is a bit complex to use. It considers each text element as a token which is then sent to NTM to receive its morpho-analysis. With this instruction, NTM does not apply its longest match strategy and simply searches its lexicons to find out whether the string is a known word.

- **Tokenizeattribute(token|chunk|dependency,"ATTRIBUTE","empty")**

This instruction is a bit complex to use. It considers each text element as a token which is then sent to NTM to receive its morpho-analysis. With this instruction, NTM does not apply its longest match strategy and simply searches its lexicons to find out whether the string is a known word. The difference with the above instruction is that it takes as input the field associated to the XML ATTRIBUTE, given in the instruction, while the other instruction uses the TEXT field of the current XML node. You can use xmlattributes(#1,vdict) to get the node attribute.

#Default

A default mode can be defined for the markup tags which are not associated with a specific instruction. The XPath must then be replaced with the keyword #default.

Example:

#default->skip(text). *//By default, any markup tag not defined in the grammar should not be analyzed.*

Example

Below is small XML document.

```
<ROOT>
        <Docid>129</Docid>
        <Title>XEROX sells parsers</Title>
        <Body id=135>
        This is an example of <Bold>a text</bold> to be analyzed.
        </Body>
</ROOT>
```

Here is a small XMLGuide to drive the analysis of that document:

XMLGuide:

```
#default->skip(text).
Docid->analyze(token).
Title->analyze(chunk).
Body->merge(dependency).

Bold->feature(boldfirst,bold).
```

Strings send to XIP

If we apply this grammar to the above document, then XIP will parse the following strings:

a) **Docid** sends the string “129” which is simply tokenized.
b) **Title** sends the string “XEROX sells parsers” which is chunked
c) **Body** sends the string “This is an example of a text to be analyzed”, with “a text” marked with *bold* and “a” marked with *boldfirst.*

Creating and Testing an XML file

XIP provides a specific set of instruction to use the structure of an XML document through the parsing process. The goal of these instructions is to use XML documents as database, to create XML documents on the fly as repositories for extracted information, and to allow one to control which XML node is under scope while parsing an XML document.

Important

These instructions are part of the Dependency Rule formalism.

XML instruction

Every XML document can be referred to in XIP with an XML pointer. An XML pointer is a string starting with a "@". Every XML instruction starts with an XML pointer.

The formalism of an xml instruction follows the pattern below:

@aliasXIP(XPath)->Method(p1,p2..,pn).

@alias is the XML pointer.
XIP XPath is an XPath with specific instruction to mangle it with XIP information.

Method is a specific method that is applied to XML nodes corresponding to the XIP XPath instruction.

XML Document Parsing XML Pointer : @_current

When an XML document is parsed with the help of an XMLGuide grammar, then this document can be referred to with a specific XML pointer: *@_current*. It is then possible to detect in the grammar which XML node is under scope and to modify the grammar behavior accordingly.

XML current node pointer: @_currentxmlnode

The last XML node which has been created or manipulated is always pointed by @_currentxmlnode. For instance, in the example below, a new XML node is created and new attributes are added to it.

Example

```
@myxml(/Root/Test)->Create();
//We add a new attribute to that node
@_currentxmlnode()->AssignAttribute("val",_val);
//We add a new XML node: FOO to that node
@_currentxmlnode(FOO)->Create();
```

XIP XPath !XIP!

A *XIP XPath* is basically an XPath expression with specific features to deal with XIP information, such as nodes or numerical variables. When the presence of specific information pertaining from XIP is needed, then this information should be enclosed with "!".

Important

There is one main difference between a XIP XPath and a XPath as defined by the W3C. The operators "AND", "OR" and "NOT" have been replaced by "&", "|" and "~".

"&" corresponds to "AND".

"|" corresponds to "OR"

"~" corresponds to NOT.

Examples:

1) /Root/Title[@lemma=!#1[lemma]!]
In the above example, we have extracted the lemma from a XIP node variable #1.

2) /Root/Title[@id=!_num!]

3) /Root/Title[@id=!_num! & @lemma=!#1[lemma]!]
In the above example, we impose a specific conjunction of argument values. The "&" corresponds to the "AND" in XPath formalism.

Instructions

We present now the list of all available instructions to deal with XML nodes.

Set(varxml)

Varxml should be a variable declared as *xml* in the variable section of the grammar. This instruction stores in *varxml* the pointer on the XML node. See *xml variables* for more information.

Example:

```
@_current()->set(varxml);
```

Create(Content)

This instruction is the main instruction to create a specific node in an XML node. The parameter of that method is optional; it corresponds to the content of the XML node that is being created. The attributes that are assigned to that new XML node are defined in the XPath instruction.

Important XML node already exists

If the *XIP XPath*, already corresponds to an existing XML node, then the *"Node Content"* is simply appended to the content of that XML node.

Example

a) |noun#1| @mydb(/root/noun[@lemma=!#1[lemma]!])->Create().

We create a new XML node, with lemma as attribute, whose value comes from a XIP node variable. This dependency rule creates for each noun an XML node with a specific lemma attribute. The content of that XML node is empty.

If we parse the sentence: *The dog is eating some soup.*

We create the following XML file for that sentence:

```
<root>
<noun lemma="dog"/>
<noun lemma="soup"/>
</root>
```

b) |verb#1| (@mydb(/root/verb)->Create(!#1[lemma]!).

In the above example, we create one single XML node, in which we store the lemma of each verb found in the sentence.

If we parse the sentence: *The dogs eats a bone and drinks some water.*
We create the following XML file:

```
<root>
<verb> “eat” “drink”</verb>
</root>
```

CreateAdd(Content,depth)

This instruction works as Create to the difference that a new XML node is created. In Create, if an XML node already exists, then the new content is merged in this node. With CreateAdd, a new node is appended to the XML tree. The *depth* parameter defines the depth at which this new node is created. The *depth* should be understood as the position of the upper node to which the new structure is appended. Typically, a value of 1 appends the new node one step above.

Example

The lady drinks and eats

```
|verb#1| (@mydb(/root/categories/verb)->CreateAdd(#1[lemma],1).
```

The above instruction creates the following structure:

```
<root>
<categories>
<verb>eat</verb>
<verb>drink</verb>
</categories>
</root>
```

If we apply this other instruction:

```
|verb#1| (@mydb(/root/categories/verb)->CreateAdd(#1[lemma],2).
```

The result is a duplication at the *category* level in the XML file:

```
<root>
<categories>
<verb>eat</verb>
</categories>
```

```
<categories>
<verb>drink</verb>
</categories>
</root>
```

- **SetNameSpace(prefix,href)**

This instruction is used to append a new namespace in a document. The user should provide the root node of the document as xpath.

Example:

```
@np(/DOC)->setnamespace("xlink","http://www.w3.org/1999/xlink") ;
```

- **SetAttribute(attribute,XIP) or Createattribute(attribute,!XIP!)**

This instruction is used to append or modify a specific XML node with a new attribute.

Example:

```
@mydoc(/root/noun)->CreateAttribute(id,!_num!);
```
We add to the XML node “noun”, the attribute id whose value is a XIP numerical variable.

- **GetText(XIP) or Assign(!XIP!)**

This instruction is used to retrieve the content of an XML node and to assign it to a XIP variable.

Example

```
<root>
<id>123</id>
</root>
```

```
@mydoc(//id)->Assign(_num) ;
```
We assign the value *123* to the XIP numerical value _num. At the end of the treatment, _num=123.

- **GetAttribute(XIP,attribute) or AssignAttribute(!XIP!,attribute)**

This instruction is used to retrieve the value of an attribute and to assign it to a XIP variable.

Example

```
<root>
</noun lemma=”chien” gender=”masc”>
</root>
```

@mydoc(//noun)->AssignAttribute(#1[gender],gender)

We assign the value of the attribute “gender” to the feature “gender” of the #1.

Getdata(dictionary)

The “getdata” instruction is used to retrieve all information about a given XML node. This information is then stored in a dictionary variable. The name of the XML node can be accessed through the key value: “#tag”, while the properties of the node are each accessed through their own attribute.

Example

If the current XML node is: <node att1=”1” att2=”2”>
Then the following instruction will retrieve:

```
Variables:
dictionary dico;

Script:

@_current()->getdata(dico);

dico[“#tag”] = “node”
dico[“att1”] = “1”
dico[“att2”] = “2”
```

InitialIndexHits(var) or IndexHits(var)

Any XML documents used as a database is automatically indexed along XPath provided from within the grammar. These XPath might contain some variables, which are automatically extracted through the indexing mechanism. *Var should be an integer.*

1) InitialIndexHits stores in *var* the number of hits that were found during the initial indexing for a specific XPath expression, before instantiation.

2) IndexHits(var) stores in *var* the number of hits which matches a specific XPath expressions with all its variables instantiated.

Example:

```
int myhits;

string data;
```

```
@db(/Root/x[@att=data])->InitialIndexHits(myhits);

@db(/Root/x[@att=data])->IndexHits(myhits);
```

If the XML database file contains 500 possible hits for this expression, then *myhits* will be 500. If *data* is set to a specific value, which matches 20 possible XML nodes, then *IndexHits* will 20 in *myhits*.

Markup(!#1!,markup_tag,att1,val1,..,attn,valn)

This instruction is used to insert mark up tags into a document according to a given phrasal or lexical node #1. This instruction can also generate as many attributes for that new mark up tag as necessary.

Important

In order to use this option in XIP, the parser should be set to "–offset" and "–nodisplay", or DISPLAY_OFFSET | DISPLAY_NONE.

Example

We have the following xml text:
<*body*>This product has been designed by Xerox in Grenoble</*body*>

We want to annotate this document with a tag "company" for all company names in that document. We use the following instruction:

```
|Noun#1[org]| { @_current()->markup(!#1!,company);}
```

This instruction reads: for a noun with the feature "org" in our chunk tree, we insert a new mark up tag "company" in our XML text as input.

The result is:

<*body*>This product has been designed by <*company*>Xerox</*company*> in Grenoble</*body*>

We can also add new attributes:

```
|Noun#1[org]| { @_current()-
>markup(!#1!,company,name,!#1[lemma]!);}
```

This instruction is the same as the previous one, however it also adds as a new attribute the lemma form of that noun. The result is:

<*body*>
This product has been designed by
<*company* **name**=”Xerox”>Xerox</*company*> in Grenoble
</*body*>

CreateDependency(dependencyName,param1,param2..)

This instruction is used to create a dependency out of an XML database file. The parameters can be syntactic nodes or attributes from the XML file.

Important: There is as many dependencies created as XML nodes matching the XPath.

Example

```
<root>
</noun lemma=”chien” gender=”masc”>
</root>

@mydoc(/root/noun)->CreateDependency(animal, lemma,gender);
```

This instruction creates the dependency: animal(chien,masc).

Test()

This method is used to check whether a certain XML node does exist.

Example

```
If (@db(/root/noun[@id=!_num!])->Test()) { _num=_num+1}
```

If the XML node /root/noun[@id=xxx] exists then the _num is incremented by one.

Compare(string | !XIP!)

We compare the text content of an XML node with “string”.

Example

```
If (@db(/root/noun[@id=!_num!])->compare(“string inside”))
{print(“match”);}
```

If the text content of that XML node matches “string inside” then we print the message: “match”.

SetText(XIP|string)

This instruction replaces the text content of the current XML node with the content of a XIP variable or a string.

Example

```
@db(/root/noun)->SetText(“This node”);
@db(/root/noun)->SetText(#1[lemma]);
```

Substring(string | !XIP!)

We check whether string is a substring of the text content of an XML node.

Example

```
If (@db(/root/noun[@id=!_num!])->substring(!_num!)) { print(“match”)}.
```

Save(filename,encoding)

This instruction is used to save an XML document which has been created on the fly in memory or which has been modified. The parameter of that instruction is a file name.

Example

1) @_current()->save(“versionbis.xml”);
The current XML file that is loaded in memory has been modified for instance new XML nodes have been created). We save a new version of that file.

2) @db()->save(“db.xml”);
We save the XML database.

For efficiency reason, a “save” should be written in a “final” clause:

```
Final:
        @db()->save(“db.xml”);
```

In this way, the file is saved only once at the end of the linguistic process.

Clean()

This instruction is used to destroy from memory a specific XML document that would have been created on the fly.

Example

a) @mydb()->Clean();

This instruction cannot be used to delete a specific XML node.

Empty XIP XPath

When a node has been detected with a given XPath, for instance after a “Test)” instruction, then XIP keeps a track of that XML node for the next XML instructions. To use this “pointer”, the linguist should simply write an empty XPath in the XPath part of the instruction.

Important

In the case that no XML node has been detected, the system uses the top XML node of the document.

Example

```
if (@db(/root/arg)->Test()) {
        @db()->Create(“value”); //we provide an empty XPath
}
```

If the “/root/arg” XML node exists, then we append the string “value” to its text field.

New Commands

Below is the list of the new commands that are available in XIP to modify the output:

-sentence

This option displays the sentence before printing the result of the analysis.

-feat

This option replaces –ne in tagger mode. It displays the internal XIP features computed for each reading after completion of the tagging stage.

-obn

The dependencies are displayed according to their left parameter Order By Node).

-ord

The dependencies are displayed according to their creation order.

-tl *number*

This option is used to split the chunk tree displayed as a text tree) along *number,* which corresponds to the maximum of characters on a single line.

-tc

This option is used to display the chunk tree as a sequence of sub-nodes

-cat

This option appends the category name of each terminal node in the chunk tree.

-nodisplay

This option suppresses any display on screen.

-xmltext depth) *filename*

This command tells XIP to parse this document as an XML file. The file can be divided in block according to the value of depth, which is optional.

-insxml

This command tells XIP to save the result of the parsing of an XML document according to the XML XIP DTD. The result is then stored in a copy of the XML document as input, as part of the XML nodes that were parsed.

-xmldb *alias filename*

This command is used to load an XML document in memory and to use is it as an external database. *Alias* is the xml pointer that would be used to refer to that document.

XIPTRANS

XIPTRANS Compiler

The compiler is part of the XIP engine and can be called from the command line:

-trans source.bse source.ar

Proprietary format, different from FST

Files are twice as large (these people are quite good, aren't they)
However, the compiler does the following thing

- Factorization
- Minimization

XIPTRANS: Lexicon files

XIPTRANS compiles lexicon files with the following format into a transducer:

- The extension is .bse (but it is not mandatory). It means *Basic Set of Expressions.*
- These lexicons contain multiple word expressions, of *one* to *n* words.
- Entries are described on three different lines:
- The first line is the surface form.
- The second line is the lemma form.
- The third line is the feature structure.
- The surface and the lemma forms can contain any characters including spaces and tabulation.

Lexicon files: Examples

Below are some examples of entries:

bag of words   #a multiword expression
bag of words
+MWE+Sg+Noun
bags of words   #the same expression in plural form
bags of words

```
+MWE+Pl+Noun
dogs                    #a single word
dog
+Pl+Noun
```

Regular expressions

The regular expressions in these files are based on the XIP regular expression format. These files are compiled on the fly.

A regular expression is threefold:
- The surface regular expression
- The lemma regular expression (which is optional)
- The feature structure, which is a list of features separated with a "+".

Regular Expression: Example

Below is the regular expression rule patterns:

If the LEMMA is not mentioned, the lemma form is the surface:
- RG_SURFACE : RG_LEMMA = +Features
- RG_SURFACE = +Features

Examples:

- "%d+[.,%,,]%d+" = "+Num+CARD". #*Recognize a number*
- "%d+[.,%,,]%d+" : %d+ = "+Num+CARD". #*Recognize a number,*

however the lemma is restricted to the first digits before the point or the comma.

Addword

It is possible to add words on the fly to memory and to store them in .bse file:

addword(alias,surface,lemma,features)

- alias is a keyword which has been defined in a "dynamic" field in a lex.trs file. If alias value is "", then the words are added in memory but not stored in a file.
- Surface is the surface form of the new entry
- Lemma is the lemma form of the new entry
- Features is a string which describes the features of the new entry. Each feature value should be separated with a "+".

Example

```
|NP#1| {
        addword(“first”,#1[surface],#1[lemma],”+Noun+Prop+NOUN”);
}
```

LEX.TRS: Keyword Fields

The different fields are the following:

- lexicons: *This field defines the list of XIPTRANS files*
- dynamic: *This field defines which files can be automatically updated with the addword command*
- onthefly: *This field defines a list of bse files which are compiled on the fly and automatically loaded as* lexicons.
- compile: *This field defines a list of bse files, which are compiled on the fly, but not added to the lexicon list.*
- vocabulary: *This field defines the list of regular expression files which are also compiled on the fly.*
- guessers: *This field defines a list of XIPTRANS files which are used to guess the POS of an unknown word.*
- guesservoc: *This field defines a list of regular expression files which are used to guess the POS of an unknown word.*
- guesstag: *When everything has failed, this field contains the string of the default category of unknown word.*
- spaces: *This field defines the list of space characters. The description is mapped over the NTM description.*
- separators: *This field defines the list of separators. The description is mapped over the NTM description*
- replacements: *This field defines the list of substring replacements.*
- equivalences: *This field defines a list of character equivalences.*

Lex.trs: Example

```
lexicons:
$TRANSFILE\numbers.ar
$TRANSFILE\english.ar

onthefly:
$TRANSFILE\toto.bse
$TRANSFILE\titi.bse

vocabulary:
$TRANSFILE\english.voc
```

```
guessers:
$TRANSFILE\open.ar

guesservoc:
$TRANSFILE\guess.voc

dynamic:
first=$TRANSFILE\first.bse
second=$TRANSFILE\second.bse

guesstag:
+guess+NOUN

spaces:
\9 \13 \32 \160

separators:
 , ; . : ! ? - _ \34 \39 ` ( ) [ ] { } ^ = / \ | * + % $ £ # < > ~ &   « » ° \10

replacements:
"aer"="fgh".
"xyz"="XYZ".
"ae" = "ssions".

equivalences:
"A" = "a".
"B" = "b".
"C" = "cç".
"D" = "d".
"E" = "e".
```

Probabilities

XIP provides some mechanisms to handle probabilities with any sorts of rules. Each rule is automatically associated with three numerical (float) values: *weight, threshold* and *value.* Each of these values can be individually instantiated from within or out of the grammar.

By default, any rule whose *value* is superior or equal to the *threshold* is triggered. The default values for each rule are the following:

a) weight=1
b) threshold=0
c) value=1

The user can also trigger a specific mechanism, which automatically computes a new *value* out of the combination of the *weight* with a *random* value. In this case, the *weight should be provided as a value between [0..1], while the threshold should be set to a value between [0..100].* The *value* which is computed by the internal randomizer will be between *[0..100].*

Important: *Value* will be computed only once for a full parse either of a string or of a text.

To trigger this mechanism the user can call XIP with the flag *–random*.

If the user utilizes the XIP API, then the following flags should be set:

Python:	XIP_RANDOM_ANALYSIS
C++:	RANDOM_ANALYSIS

(This flag corresponds to the binary value: *274877906944*)

Instructions

XIP provides instructions to handle these different values at three different levels: from within, with specific scripts instructions, or from C++ and Python API.
At the end of parse, it is possible to *store* for each rule, the different *weights, thresholds or values that have been computed in a file.* This file can be reloaded anytime as an *addendum* or *parameter* file. For instance, an *Addendum* section in a GRM file can store the path of a probability file.

The instructions are the following:

Script Instructions

In the script language these values can be handled in the following way:

setruleweight(id,w):	set the weight to rule id
setrulethreshold(id,t):	set the threshold to rule id
setrulevalue(id,v):	set the trigger value to rule id

getruleweight(id):	return weight of rule id
getrulethreshold(id):	return threshold of rule id
getrulevalue(id):	return trigger value of rule id

In all the above rules, *id* is the unique identifier of a rule. To use this identifier, it is possible to loop between rules. In this case, one can use the following instructions:

@nbrules:	return the number of rules in the grammar
rulecounter(id):	return the number of occurrence of rule id
rulelayer(id):	return the layer number of rule id
ruletype(id):	return the type of rule id
ruletypestring(id):	return the type of rule id as a string

With these instructions, it is possible to focus on specific rules, which have either a specific type or belong to a specific layer. The number of occurrence is automatically set when a document is parsed. Each rule application increments its internal counter.

Once a document has been parsed, one can store the computed values with the following instructions:

saveprobabilities(f):	store the probabilities in a file f
loadprobabilities(f):	load the probabilities from file f

As said below, a *probability file* is very similar to a regular XIP file. It has a specific field name: *probabilities*. Data are organized in the following way for each rule:

IDrule^{i}=weight^{i}:threshold^{i}:value^{i}*counter^{i};IDrule^{i+1}=weight^{i+1}:threshold^{i+1}:value^{i+1}*counter^{i+1}*etc…*

These structures are all concatenated one to the others.
The best practice is to call *saveprobabilities* from within a *FINAL* section.

The initial probabilities can be either set from the API or reloaded from a probability file, which can be loaded from within the grammar or using a script, C++ or Python instruction.

C++ API

XIP C++ API provides the same functionalities as the script language but at a different level. Each script language has its C++ implementation. The main difference is that the grammar handle should be provided for each call to these functions.

```
Exportation char XipSetWeight(int ipar,int id,float v);
Exportation char XipSetThreshold(int ipar,int id,float v);
Exportation char XipSetValue(int ipar,int id,float v);
Exportation float XipGetWeight(int ipar,int id);
Exportation float XipGetThreshold(int ipar,int id);
Exportation float XipGetValue(int ipar,int id);
Exportation int XipRuleCounter(int ipar,int id);
Exportation int XipRuleType(int ipar,int id);
Exportation string XipRuleTypeString(int ipar,int id);
Exportation int XipNbRules(int ipar);
Exportation int XipRuleLayer(int ipar,int id);
Exportation void StoreProbabilisticModel(int ipar,string filename);
Exportation char LoadProbabilisticModel(int ipar,string filename);
```

Furthermore, the *randomize action* can be set with XipSetDisplayMode using the value: RANDOM_ANALYSIS.

Generation

XIP offers a complete system of rules to generate new texts from a set of dependencies. The generation rules are very similar to dependency rules with one strong difference: they produce trees instead of dependencies.
The nodes in these trees must be declared in a specific NODES section.

NODES (French: Noeuds)

The NODES section is the section where the generation nodes must be declared. The declaration is the same as in a FUNCTIONS section.

Example:

```
NODES:
GNP[gnp=+],
GP,
GVP.
```

These nodes will be both handled by the generation rules as a node in a tree and as a function.

TOKEN, STOKEN

TOKEN and STOKEN are two nodes which are pre-declared. They are used at the end of the process, when the generation tree is traversed to generate either a new surface form with a look down in an FST file, or by displaying the surface form, if this form has already been provided.

TOKEN *is used in conjunction with the GENERATEFEATURES section to compute the surface form of word, given its lemma and its features*

STOKEN *utilizes the surface form of the syntactic node as input.*

Rules

The generation rules are very similar to dependency rules as they use as input both dependencies and nodes. However, the result of a generation rule is a tree structure, which is, at the end, traversed in depth first to generate the final sentence.

The pattern of a GENERATION rule is the following:

If (test) NX{NY{..},…}.

The test is a conjunction or a disjunction of both dependencies, nodes similar to a dependency rule, while the output is a tree structure whose depth is recursively described with "{…}", similar in this respect to a reshuffling rule.

NX, NY must have been declared in a NODES section.

Important:

A layer number can be specified for this rule. If none is provided, then this rule is automatically executed at the end of each generation layer.

Declaring a rule: (generation or génération)

A generation rule is declared in a GENERATION section.

Variables: $#...

The formalism of a generation rule provides a specific variable operator: $#x, where x is any number, which can be used to refer to a specific node. This operator can be used to rebuild a generation tree locally.

The "^" operator

This operator in the case of a generation rule is used to indicate which is the root in the generation tree under which new nodes will be appended. When this operator is used, new NODES are appended at the end of its existing siblings. The order in which these NODES are appended can be controlled with ORDER rules.

Example

Basic Rule

Generation:

1> if (subject(#1,#2)) GP(#1){GNP(#2){GN(#2)},GVP(#1){GV(#1)}}

If we apply this rule to the following example:

Subject(drink,dog).

The system will generate:

```
                  GP(boire)
   +---------------------------+
   |                       |
GNP(dog)              GVP(drink)
   |                       |
 GN(dog)             GV(drink)
```

Adding a NODE

1> if (^GNP(#1)) gdet(det[lemme="the"]).

This rule adds a determiner to a GNP node. In this case, as this determiner does not exist, it is created on the fly with a specific lemma. However, the GDET node is placed at the end of the GNP siblings.

```
                  GP(boire)
   +---------------------------+
   |                       |
GNP(dog)              GVP(drink)
   |                       |
 +----+               GV(drink)
      |          |

 GN(dog)GDET(the)
```

We need a specific sort of rules to avoid this error.

Using a Variable

1> if (GNP$#1(#1)) $#1{ gdet(det[lemme="the"])}.

This rule is similar to the previous one to one difference. In this case, we use a variable $#1 to build the sub-tree. Furthermore, in this case, the variable must be used in the result structure.

Absolute position: [$=0|1|-1]

The other way to position a new NODE in a generation tree consists of using the "$" operator as a feature associated to a new NODE. "$" is treated as a feature, which is deleted once the NODE has been both created and inserted at the right position. This operator takes three values:

a) $=0 The NODE is inserted at position 0 in the NODE sequence

b) $=1 The NODE is inserted at position 1 in the NODE sequence

c) $=-1 The NODE is inserted right before the last NODE in the NODE sequence.

Example:

If we use the following rule with: DETERMINER(dog,the)

If (determiner(#1,#2) & ^NP(#1)) det[$=0](#2)

Then, the new NODE *det* will be inserted as the first element of NP.

Testing a sub-tree in an IF

It is possible to test within an IF, a complete tree pattern over a NODE tree. This test uses exactly the same pattern structure as the one which is used to create a subtree, however in a IF:

If (S$#1(#1) & $#1{NP$#2(#2),VP$#3(#1)} …) …

After application of such a pattern, the different variables $#2,$#3 will be instantiated. The structure itself is not deterministic. It will apply to as many nodes as possible.

Important:

The tree pattern cannot contain any Kleene operator. However, it works in very specific way, which is very different from the other regular expressions in XIP. The expression: NP,VP, will match any *sub-tree containing in this order NP and VP, whatever NODES might be in between.* In other words, this expression is equivalent to: NP,?*,VP.

ORDER Rules (French: ordre)

The ORDER rules are used to control how a NODE can be appended under a specific NODE.

NODE[features] < NODE[features].

[features] < [features].

Important:

A layer number can be specified to these rules. If none is provided, then these rules are automatically executed for each generation layer.

The role of these rules is to control the position of any new nodes within a tree.

Example:

For instance, in the previous rule, the determiner was added at the end of the NODE’s siblings, which in English, is not the right position. A simple rule such as:

1> GDET < GN.

will ensure that the determiner will be placed before the nominal NODE.

Functions

XIP provides some functions to deal with the different nodes:

- NODEMOTHER($#1,$#2), *this function tests or returns the mother node of $#1 in the GENERATION Tree.*
- NODENEXT($#1,$#2), *this function tests or returns the next node of $#1 in the GENERATION Tree.*
- NODEPREVIOUS($#1,$#2), *this function tests or returns the previous node of $#1 in the GENERATION Tree.*
- NODEDESCENDANT($#1,$#2), *this function tests if $#2 is the descendant of $#1.*
- NODEDAUGHTER($#1,$#2), *this function tests or returns the first daughter node of $#1 in the GENERATION Tree.*
- NODELAST($#1,$#2), *this function tests or returns the last daughter node of $#1 in the GENERATION Tree.*

Lookdown

XIP also provides a specific formalism to give a word its proper form. This operation called *Lookdown* is based on an external lexicon which must be provided by the linguist in a specific section of the GRM file. A *Look Down* consists in providing a FST with both a lemma and a string of features, the system returns then the corresponding *surface* form.

Example:
The word *chien* together with the feature string *+Masc+PL+Noun,* returns *chiens.* The same lemma *chien* with the feature string *Fem+SG+Noun*, return the string: *chienne.*

GRM Section: Generation

This section in the GRM file is used to store the pathname of a FST script file, which contains the pathnames of the files used to apply a lookdown.

GenerateFeature (French: GénèreTraits)

This section has two goals:

- It provides a list of feature strings which will be used to return a given surface form.
- It provides a list of feature string, which will be used to *update* the feature of a given node.

A *GenerateFeature* entry is two-folds: on one hand, a specific category and feature structure *à la XIP,*on the other hand, a string of features separated by a '+' as they appear in the FST file.

Example:

//Regular lines. The system tests each left-hand part on the node, then buils a string combining the lemma form and the string on the right side of the rule. If the lookdown succeeds, then the appropriate surface form is returned, and the system stops.

verb[tense:pres,pers:2] =+IndP+SG+P2+Verb.
det[gender=fem,pronsubj:~] = +Fem+SG+Def+Det.

//In the following example, not only is the surface form computed, it also updates some features on the node.

noun[gender=masc,pronsubj:~] = +Masc+SG+Noun.
noun[gender=fem,pronsubj:~] = +Fem+SG+Noun.

Triggering a Surface Computing

The above rules are called in only two cases:

- When a TOKEN node is encountered while building the final sentence at the end of the generation process.
- If a *INITLOOKDOWN*(#1) is applied on the node #1 in a generation rule. In this last example, the lexical #1 node can be automatically updated with specific features instantiated by a GENERATEFEATURE rule.
    - **Example:**

    If (subject(#1,#2) & initlookdown(#2) & #1[agreement]={#2})
    ~.

If a noun's features are unknown, then INITLOOKDOWN can be used to update them, and then pass these features to the verb node.

Modifying Sentence Generation

It is possible to have access to the final string, which then can be modified according to lexical rules. For instance, in English, the determiner "a" is replaced with "an", if the next word starts with a vowel. XIP provides then a few functions, which can take these cases into account, namely: *words*, *setword* and *getword*. In this context, a *word* is the final surface form that was produced at the end of the generation process. Hence, these rules should be applied at the very end of the grammar to avoid side effects.

Words("str1","str2",…,"strn"): this function takes as input as many regular expressions as necessary, which will then be tested in sequence. Each string will be tested against a specific *word* generated in the sentence.

Setword(i,wrd): this function modifies a *word* in the sentence, whose position *i* corresponds to the input position in the *words* instruction. As a *words* can apply to more than one element in the sentence, this *setword* modifies *all* instances that were detected by the previous instruction.

Getword(i,wrds[]): this function returns the list of *words*, whose position *i* corresponds to the input position in the *words* instruction. The return variable is a vector of strings, as more than one string could be found at this position.

Example

If (words("a", "[a,e,i,o,u]?+") & setword(0, "an")) ~.

If a word starting with a vowel is preceded by "a", then we replace "a" with "an". 0 here is the position of the test on "a" in *words.*

More functions

XIP provides two more functions, which benefit from the generation FST file:

- nblemma=lookup(wrd,lemfeat), *where* wrd *is a surface form. It returns a list of lemma + fetures strings in the* lemfeat *string vector. This function also returns the number of lemmas found.*

- res=lookdown(lemma,feats), *this function returns the surface form corresponding to a* lemma *and a* feature string.

GRM sections

The GRM file is the most important file in a grammar. It is the one which handles the different files needed for a given grammar. We present in this section the different fields, which are available in this file.

- grm: Use to load another grm file within the current grm file
- path: To declare a new environment variable. The next line should contain: variablename=a path...
- data: To declare files which are necessary for the grammar but not loaded by XIP.
- generation: The FST lexicon for look down.
- grammar: The grammar file
- parameter *or* addendum: Adding grammar files

- strategy: values are: BREADTH or DEPTH
- lookup: The lookup file
- flags: Some FST flags
- tokenizer: The tokenization file

- ntm: The NTM script file

- hmm: The HMM files

- crfmodel: The crf model file
- crfmkcls: The CLS file
- crfbrown: The Brown classification file

- trans: The lexicon are stored as TRANSDUCER files.

- blockingrulespace: Rule space that is blocked
- variable: To declare a variable, which will be used in ifdef sections.

- indexingbuffersize: The buffer size to index semantic rules.
- number: This value defines the bit vector size used to store features.
- conversion: value is *utf2ascii* to enable utf8 conversion
- layer: this value defines the first layer number in the grammar