XLFG Documentation

version 9.3.3 – october 2014

Lionel Clément

Contents

Acknowledgment

I would like to thank the following people for their valuable contributions to different parts of XLFG (documentation, graphic design, contributions to the implementation of the front-end, samples, feedback, etc.)

Olivier Bonami (Univ. Paris-Sorbonne)

Kim Gerdes (Sorbonne Nouvelle, Paris)

Hélios Hildt (Univ. Bordeaux Montaigne)

Fiammetta Namer (Univ. Lorraine)

Hugo Nioteau (Univ. Bordeaux)

Pape Ousmane Sow (Univ. Bordeaux)

Disclaimer

This document is neither an introduction to Lexical-Functional Grammar (LFG) nor a manual on formal linguistics. It is intended as a resource for people who want to use XLFG for a class, for research purposes, or out of mere curiosity. Readers who wish to learn more about LFG are kindly asked to consult the reference section at the end of this document.

XLFG is constantly evolving; thus, parts of this guide may well become obsolete in the future. This document was meant to be used with XLFG version 9.3.3. It will be updated and made available online each time XLFG is modified.

XLFG has been designed to allow the conservation of works for multiple users (students, teachers, lecturers, researchers, or anybody else). Nevertheless, it is not designed to backup data. We disclaim any liability for the loss of work. To make sure work and data is properly saved, a variety of alternative methods are readily available.

XLFG is often used for exams and tutorials. While we endeavor to do the utmost to prevent server interruptions, we disclaim any liability for problems resulting from the use of this Web site during exam or tutorial sessions.

XLFG has been designed to with relevance to the field of linguistics. The comments and opinions in the sentences or comments are those of the author alone and do not necessarily reflect those of the XLFG operator.

We reserve the right to terminate any account and remove all contents associated with it, at any moment.

Commitments

We commit to not communicate, for any purpose whatsoever, the email addresses used to register with XLFG, or any personal data.

We commit to removing any account at the request by email of the person concerned and do not keep personal data and records.

1 What is XLFG?

XLFG is a fast, accurate deep parser for LFG grammar. These outputs are phrase structures, predicate-argument structures and predicate-thematic relations. It allows any user to parse an extensive (in fact unlimited) set of phrases in a long sentence with a realistic lexicon in just a few hundreths of a second.

XLFG is divided into two separate parts: back-end and front-end.

For those interested in the back-end part as yet unavailable on the Web-portal, design for Natural Language Processing applications, please contact the author Lionel Clément.

This present documentation concerns more particularly the XLFG's front-end, designed to experiment LFG grammar online with small lexicons.

The XLFG's front-end will hereafter be referred to simply as "XLFG".

2 Installing XLFG

XLFG (front-end Web-portal)

XLFG does not require software installation. XLFG may be used as an online service using a web browser from any computer connected to the Internet.

One only has to go to the following page: `http://www.xlfg.org/` with the Browser one prefers.

Nevertheless, XLFG is compatible with web browsers that can display MathML, SVG et XHTML. It has been tested on Internet Explorer, Firefox, Google Chrome, Safari and Opera.

XLFG's back-end

The back-end of XLFG has been is developed in C++. The usual command to install it on a `/usr/local` directory is:

```
(cd xlfg; ./configure ; make ; sudo make install)
```

Please refer to the content of `INSTALL` for details on back-end installation.

3 Accounts

Checking in is not mandatory for using XLFG. However, registration allows users to share their work and save it on the server.

Creating an account is fast, free and easy. Just sign-up by completing a form, including a username, an email address and a password.

To complete the signup process, we need to confirm that the email address you're using is your own: a confirmation by email will be sent to you to confirm your registration for the website.

Workshops

Any recorded user may create workshops. Workshops are designed for organising classes. A teacher or a lecturer may create a workshop, projects related to it and may then assign rights for these projects according to the individual member whose part of the project is editable or not.

The workshop owner adds members that are XLFG recorded users identified by their logins.

Projects

A project is a form with a title, an abstract, input and output parameters, grammar and lexicon.

The owner may assign each project to one of their workshops and then accord privileges for each part of the project: two levels of privilege are available for any workshop member: Read-only access, Read-write access. Obviously, any user may build a project without attaching it to his or her workshops. In any case, XLFG grants privileged access to the user's own projects.

When the same project is shared by a workshop, only copies of it are editable by each member. Thus, the workshop owner may supervise a project and read or correct the different changes introduced by each member to the original.

A practical example of using projects and workshops would be to organise a tutorial:

The teacher creates a workshop and invites the students to enroll. Then he produces a grammar with read-only privilege and a lexicon with read-write privilege for our example. He adds some private comments without read-only privilege and public ones with read-only privilege.

Each student reads the project and may change the lexicon entries by following the teacher's recommendations without any possibility of changing anything else.

The teacher may read each result and add some personal comments for himself.

Privilege	Workshop member's access	Owner
none	not visible	shared data modifiable
read-only	visible but not editable	shared data modifiable
read-write	modifiable	individual data modifiable

Figure 1: Here is a summary of privilege access

4 First steps with XLFG

4.1 Using one of the sample grammars

The easiest way to get to know XLFG is to use one of the sample grammars. Choose *English with functional-structure* in the *Samples* menu. A page opens with a number of pre-filled fields.

Input contains the sequence that will be parsed: *John sees the man with a telescope* in our first example.

Grammar contains the syntaxic rules, written as context-free rules allowing to build phrase structure trees, plus functional constraints allowing to build functional-structures.

Lexicon contains the lexicon, written as a sequence of words followed by the relevant terminal symbol from the grammar and a functional structure associated with it.

Output contains a list of buttons to choose the display mode you prefer. For now keep this standard configuration unchanged.

When clicking on the *Parse* button, a new window (or a new tab depending on the configuration of your browser) opens, which contains:

- A graph encoding the lattice of words sequence and all the constituents found (fig. 2).
- One of the two possible constituent-structures for the sentence (fig. 3).
- The corresponding functional-structure for the sentence (fig. 4).

If this does not work, perhaps the browser used has been configured to disallow popup windows or new tabs. Uncheck this option.

In addition, this new window shows the version number of XLFG that was used on the server, the time it took for the analysis to go through (in seconds), and the number of analyses that have been found.

To see the other constituent-structures and the related functional-structure, click on the link labelled `next` to the `VP` node. This link brings the other possible analyses of the verb phrase (in which the phrase *with a telescope* is attached to *the man* and is not a complement of the verb *sees*).

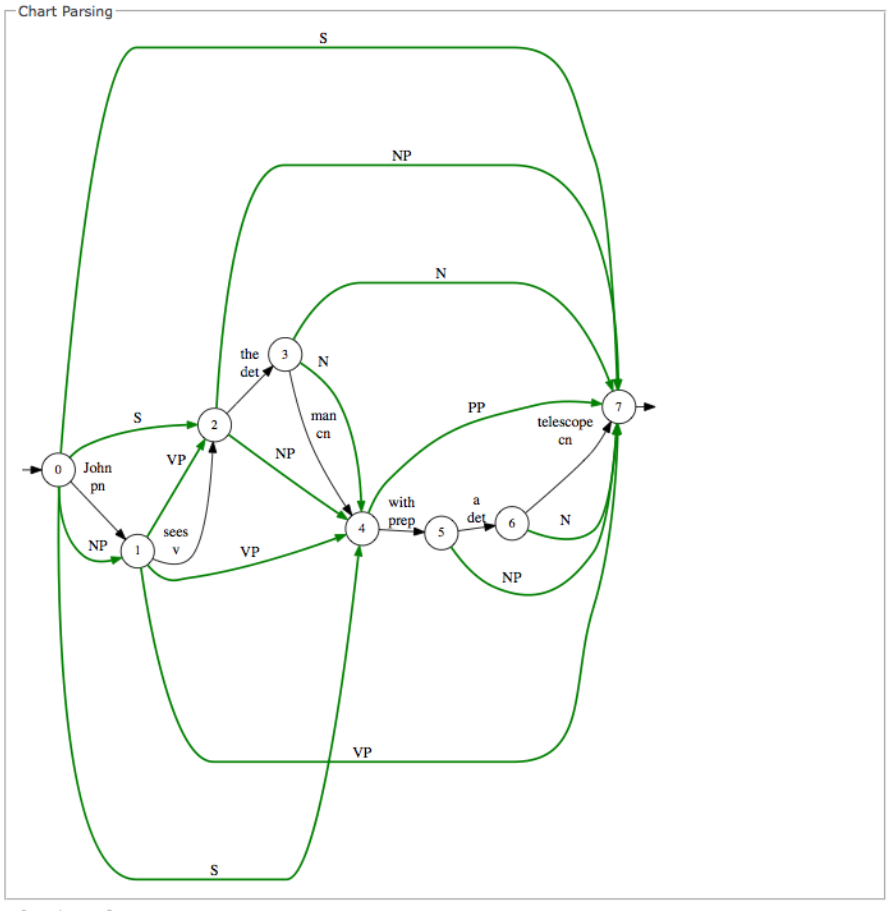

Figure 2: Lattice of words and phrases

Click on the other links of the tree to see the related functional-structures.

As this number can be quite large or unlimited, XLFG displays only one tree at a time.

4.2 Changing the example

If you are logged on, you may now try to input other examples and experiment with different display modes and parameters: *John sees the man*, *the man runs*, etc.

Notice that capitalization and punctuation are taken into account; one should be carefull about this when writing a grammar.

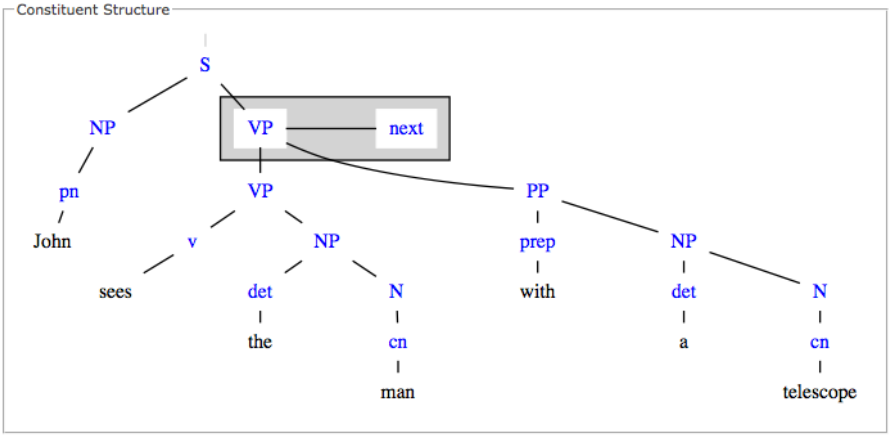

Figure 3: Constituent-Structure

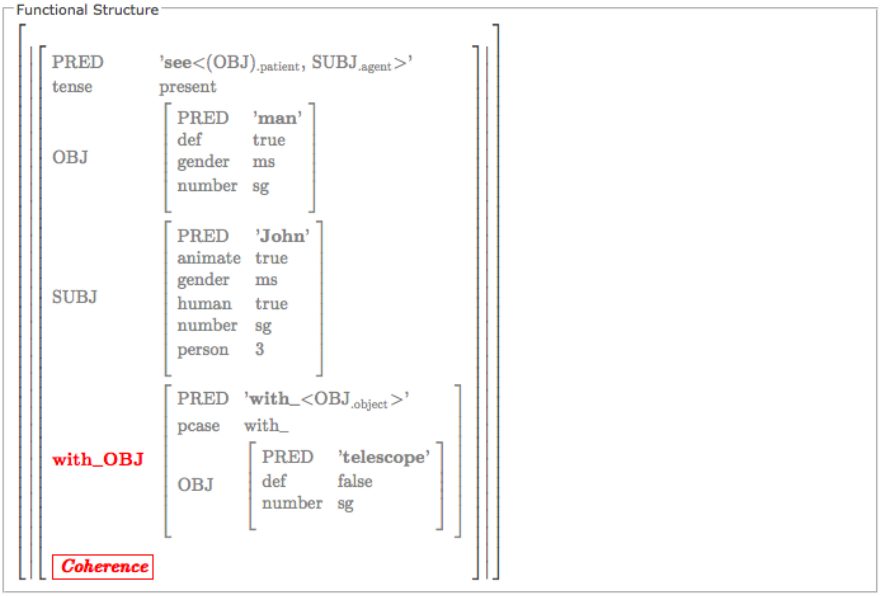

Figure 4: Functional-Structure

4.3 Error messages

If you make a mistake, you will get a (hopefully helpful) error message.

4.4 Modifying the grammar

You may write a new grammar from scratch or modify one of the given projects. In both cases just add rules conforming to the XLFG conventions.

For instance, adding the following rule to our example:

```
N → Adj N
{
    ↑ = ↓1;
    ↓2 ∈ (↑ MOD);
};
```

and the lexical entry:

```
little Adj[PRED:'little'];
```

will allow you to parse the sentence *John sees the little man with a telescope.*

At this stage we need to give a more precise description of XLFG grammars, although the sample grammars should provide an excellent idea of what they look like and allow data to be manipulated. The following sections describe the makeup of an XLFG grammar in detail. We start by explaining why grammars found in LFG books cannot be implemented directly into XLFG in some cases.

5 LFG theory and XLFG parsing

5.1 Solving equations

XLFG implements a parser for LFG grammars whose main feature is to rely on a shared representation for different parses of the same sentence. For instance, the sentence *John sees the man with a telescope* has two parses, where the phrase *with a telescope* either modifies the noun *man* or is a complement of the verb *sees*. In both cases, the phrase *with a telescope* is the same. It is thus convenient to share this part of the two parses. This sharing avoids a lot of redundant computations and makes XLFG quite efficient, even when used with realistic grammars and lexicons based on real data.

Whereas the `PP` is the same in both parses of the sentence, its function in the sentence is not the same. The structures it attaches to are different and cannot be shared between the two parses.

In LFG, the rule describing `PPs` modifying a noun might be written as:

$$\begin{array}{llll} \text{NP} \rightarrow & \text{Det} & \text{N} & [\text{PP}] \\ & \uparrow = \downarrow & \uparrow = \downarrow & \downarrow \in (\uparrow \text{ADJ}) \end{array}$$

In this rule, the f-structure of the noun and noun phrase are identified through the use of equality. This structure also contains a set-valued feature `ADJ` whose value must contain the f-structure of the modifier (in our example, *with a telescope*). It is evident that such a statement cannot be verified when the noun has no complement, because the `PP` attaches outside of the `NP`. Thus, in LFG, a separate computation must be made for each parse: each phrase structure tree requires its own f-structure computation. Without such a condition, the system of equations would have no globally coherent solution.

Obviously, an exponential number of trees are produced in case of structural ambiguity, unlimited in the case of cyclic derivation (in some grammars with empty categories for exemple). In XLFG, we decided not to limit our analysis to a small number of trees on the one hand, and we would like to remain the semantic of equality on the other. Thus, we chose to have shared phrase structure representations, in order to be able to parse highly ambiguous sentences. For this reason a different solution for computing f-structures needed to be built which did not affect either the LFG linguistic theory or the associated formal model too greatly.

XLFG models the Φ projection (from constituent structures to functional structures) using copy operations rather than strict unification for the solution of equations.

This design strategy explains why some LFG grammars cannot be used directly in XLFG, but must be adapted to meet conventions and constraints that are particular to XLFG.

This is the XLFG version of the preceding rule:

```
NP → Det N [PP]
{
    ↑ = ↓1;
    ↑ = ↓2;
    ↓3 ∈ (↑ ADJ);
};
```

In this declaration, the values ↑, ↓1 and ↓2, which correspond respectively to the f-structure of the NP, the determiner and the noun, are not identical during parsing. The '=' symbol which denotes an equality of its operands is implemented in XLFG as a bottom-up copy operation.

The declaration ↓3 ∈ (↑ ADJ) modifies the f-structure of the NP, but it must not modify the f-structure of the noun, because that f-structure is shared in the representation of both parses of the sentence.

In our example, the value of the f-structure ↑ is the unification of ↓1 and ↓2 to which the set-valued feature ADJ is added; but the values of ↓1 and ↓2 are not modified.

Parsing in LFG thus relies on a general rule that f-structures propagate bottom-up, from the most embedded to the least embedded, through copy operations.

As this exemple attests, XLFG notations are close to those commonly used in LFG literature and a typical user should not be too greatly perturbed.

Implementation

Given a shared forest where N_i dominates N_j, an equality declaration x `=` y where x is embedded in a projection of N_i and y is embedded in a projection of N_j.

Given $\phi \neq \bot$, the most general unifier (mgu) for x and y. x is assigned with $\phi(x)$.

Hereafter, the term *assignment after unification* will be used to refer to this implementation.

6 Using the XLFG Web-portal interface

XLFG Web-portal interface is pleasant to use and extremely intuitive and simple.

All you need to do is connect onto the interface from any computer using any browser. Once you have accessed the XLFG software, just let yourself be guided by the XLFG interface.

If you want to try XLFG with samples only, you do not need to create an account. If, however, you do wish to create new projects and save your work on our server, you must create an account by clicking on Sign-up in the top right corner. To log into an existing account, click on Login in the top right corner.

Once logged in, the XLFG Web-portal interface looks like this:

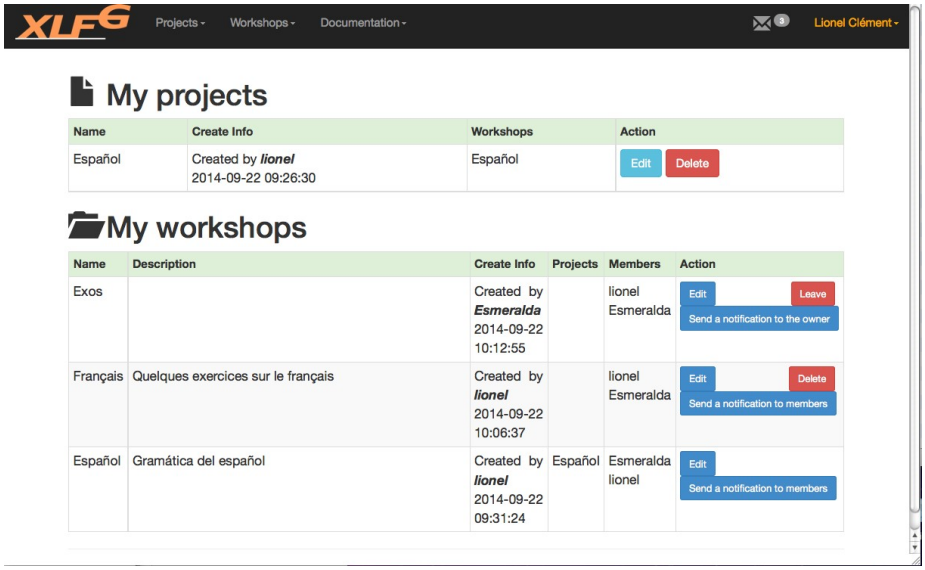

Figure 5: Home XLFG Web-portal

Workshops

If you are the administrator of a workshop, you can click on the button `Edit` to add users and projects. You will then be able to create and access the various projects of the workshop to which you belong. Otherwise, you will only be able to see information concerning the workshop and access shared projects in workshops of which you are a member.

Projects

Each user may edit and save projects involving any input to parse, a grammar or lexicon and parameters are displayed.

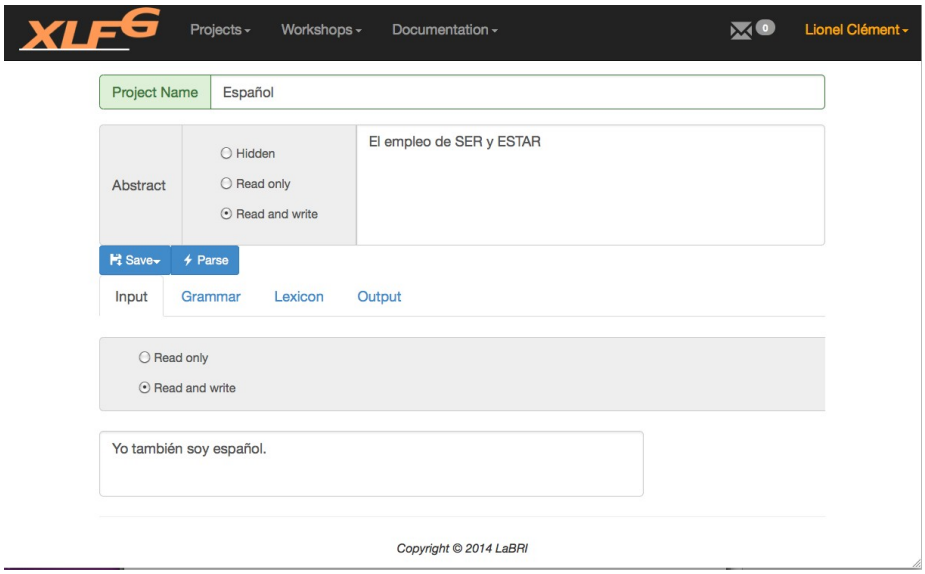

Figure 6: Project page

You can now share your projects with the other members of a workshop which you administrate.

Notifications

Any recorded user may send notifications to the workshop administrators, the entire team, or to the members of a given workshop he or she administrates. The *unread notifications* icon then appears at the top of the XLFG home-page. By clicking on it, new notifications may be read.

Notifications must only be made for the purpose of managing XLFG workshops, XLFG projects, or communicating XLFG data and results. Using notifications for any other purpose may cause the user to be banned from using the XLFG service.

7 XLFG grammar

An XLFG grammar consists of a set of phrase structure rules.

Phrase structure rules describe clause structure, or more precisely regularities in clause structure. Constituents of the clause differ in category (noun phrase, verb phrase, etc.), in position (pre-verbal, post-verbal, etc.) and in grammatical relations (or syntactic functions) they enter into (subject, object, etc.).

Categories for phrases are defined by the projection of a lexical element (a noun, verb, etc.) and a constitution.

Syntactic positions are determined by word order and the structural arrangement of words.

Syntactic functions may be expressed by case morphology (case endings, typically in "free word order" languages), or by lexically-registered valence requirements.

The aim of an XLFG grammar is to use theoretical constructs from LFG to describe the syntactic regularities of a language. XLFG will produce a parse for grammatical sentences. The analysis of ungrammatical sentences will display information allowing the user to determine which constraints were violated. These constraints are always typeset in red.

The following examples are not meant to be part of a realistic grammar fragment, but to illustrate various theoretical concepts.

7.1 XLFG Notations

The content of a XLFG grammar and lexicon is made with UTF-8 encoding characters. Capitalizations are taken into account to distinguish notations. Blank characters and line feeds are not taken into account, with the exception of strings.

Comments

Comments are destined for the reader only and are skipped by XLFG. A one line comment start with the "//" symbol, the text is skipped up to the end of the line. A multi-line comment starts with "/*" and ends with "*/". The enclosed text is skipped.

Identifier

Whitout any quotes, all the strings beginning with a latin character, or "_" followed by alpha-numeric characters are identifiers, when they are not a keyword.

- The latin characters are a ... z, A ... Z, à, á, â, ã, ä, å, æ, ç, è, é, ê, ë, ì, í, î, ï, ð, ñ, ò, ó, ô, õ, ö, ø, ù, ú, û, ü, ý, ÿ, þ, À, Á, Â, Ã, Ä, Å, Æ, Ç, È, É, Ë, Ì, Í, Î, Ï, Ð, Ñ, Ò, Ó, Ô, Õ, Ö, Ø, Ù, Ú, Û, Ü, Ý, Ÿ, Þ, ß

- The keywords are the following: `PRED`, `LEXEME _INTEGER_`, `_UNKNOWN_`, `_REAL_`, `_TEXT_`, `functions`, `startSymbol`, `not`, `if`, `else`, `with`, `in`, `switch`, `case`.

In order to write identifiers without latin characters, one can use " ‘ " quotes. For exemple the rule `S` → `Subject Object Verb`; is written in Tibetan characters:

```
`ཚིག་སྒྲུབ་` → `བྱེད་པ་པོ་` `བྱ་བའི་ཡུལ་` `བྱ་ཚིག་`;
```

Symbol

An identifier is a symbol used on different occasions:

- As a term in grammatical rules (`S`, `NP`, `DP`, ...).
- As an attribute name or atomic value in feature structures (`number`, `singular`, ...).
- As a grammatical function or a thematic role name (`Subject`, `agent`, etc.).

The maximal number of different symbols in the same grammar is limited to 120.

Constant

Constants are used to encode an unlimited list of terms (lexemes, forms, etc.).

Constants are written with simple quotes.

String

Any one line sequence between two " characters is a string. In order to design the double quote itself, one may use the escape character "\".

Strings are used to encode lexical forms.

7.2 Phrase structure rules

This is an exemple of XLFG phrase structure rule:

```
VP → [aux] V [NP|S];
```

Brackets indicate optional constituents, the vertical bar indicates an alternative between `NP` and `S` .

This rule describes the composition of a phrase of type `VP`: this phrase consists of a possible constituent `aux` followed by `V` and possibly `NP` or `S`. These three constituents must be contiguous and in the specified order.

In order to reiterate a constituent in a rule, one has to write a recursive rule instead; because the usual LFG Kleene star notation is not allowed in XLFG. The folowing LFG rule:

VP $\rightarrow$ V PP*

is an equivalent to the XLFG one:

```
VP → V PPs;
PPs → PP PPs;
```

Rules may be recursive to the left or to the right, immediately or not. This is used to model recursive phrase embedding, as e.g. in `NPs` occurring inside relative clauses themselves occurring inside `NPs`.

```
N → N AP;
N → AP N;
N → N RelVP;
RelVP → RelPro S;
S → NP VP;
NP → Det N [PP];
...
```

A given constituent type need not have a unique possible composition. Alternate compositions are described by multiple rules with the same left hand part.

```
VP → [aux] V [NP | S];
VP → VP adv;
```

A phrase structure rule may have an empty right hand side. This allows for an explicit modelling of empty categories, as are postulated in some syntactic frameworks. This is particularly relevant when such rules are associated with functional descriptions giving rise to constraints on grammatical functions.

```
NPro →;
```

Start symbol

The start symbol corresponds to the type of phrase XLFG will attempt to parse. It is most often of the type Sentence, but of course XLFG can be used to parse any type of phrase compatible with the grammar.

By default XLFG assumes that the first symbol it encounters in the grammar is the start symbol. If this behavior is not appropriate a different symbol may be specified by writing the given statement using the keyword `startSymbol`:

```
startSymbol: NP;
```

7.3 Functional structures

A functional structure (*F-structure* hereafter) is represented as a feature-structure, namely a set of attribute-value pairs. It is represented with brackets, the features are separated with commas and the attribute-values pairs with the colon character:

```
[PRED: 'snore<SUBJ>',
 TENSE: present, MOOD: indicative
 SUB: [PRED: 'John', NUMBER: sg, person: 3]]
```

While an attribute is an identifier, the possible values for attributes are atoms, embedded F-structure, or a set of embedded F-structures.

- The atomic values are:
  - A list of possible values separated by the symbol "|". This value subsumes any sub-list.

```
MOOD: indicative | subjunctive
```

– A string written with quotation mark ”

```
FORM: "that"
```

– A value for a predicate that is a lexeme, grammatical functions and thematic arguments.
The value of a PRED is written between two single quotes. A regular lexeme is noted with a single symbol. A prefix lexeme (resp. suffix lexeme) is written with a "-” after it (resp. in front of it).
Following LFG conventions, the grammatical function list is noted between chevrons for those which correspond to a thematic argument, and after for those which do not.
The grammatical functions are noted with identifiers separated with commas, and the corresponding thematic roles are noted by an identifier after a dot character.
See below (paragraph 7.3) for details on PRED attribute.

```
PRED: 'SNORE<Subj.agent>'
PRED: 'SEEM<XCOMP.theme>Subj'
```

- A set of F-structures is written within braces:

```
MOD: {[PRED: 'little'], [PRED: 'big']}
```

The attribute PRED

This attribute is central to the analysis of an utterance. The `PRED` feature of an f-structure is projected from lexical entry — for instance a particular reading of a polysemous lexeme —.

`XLFG` lexica are simple, and do not allow us to apply lexical rules, nor α projections. We assume the lexica used for XLFG come from time-deferred applications not included in XLFG software.

However, the user can associate grammatical functions with particular arguments through PRED specifications and assign them thematic roles as requested in LFG theory.

Here is a sample PRED feature in the XLFG notation:

```
PRED: 'place<SUBJ.agent, OBJ.patient, OBL.location>'
```

In this example the lexeme is *place*. It combines with three arguments corresponding to the three functions `SUBJ`, `OBJ`, `OBL`. Each one is associated with a thematic role: respectively `agent`, `patient`, and `location`.

Functions that do not instantiate a thematic argument of the predicate should be listed outside the angled brackets. This is the case e.g. for the impersonal subject of weather verbs such as *rain*:

```
PRED: 'rain<>SUBJ'
```

or for subjects of raising verbs such as *seem*:

```
PRED: 'seem<XCOMP.theme>SUBJ'
```

In other cases, a single function may correspond to different arguments of the predicate. For instance in the following examples, the subject of *crash* corresponds either to the agent or the patient argument.

- The computer crashed.
- Luke crashed the computer.

In XLFG one may explicitly annotate a function with the name of the role of the argument it realizes.

```
PRED: 'crash<SUBJ.patient>'
PRED: 'crash<SUBJ.agent, OBJ.patient >'
```

The links constructed by such argumental relations are displayed as a dependency graph. This graph may serve as a first step towards a semantic representation.

In order to describe a lexeme attribute in functional descriptions, one may use the keyword `LEXEME` on paths: (f LEXEME) stands for the lexeme of f F-structure.

Example:

```
(↑ LEXEME) = seem;
↑ = [PRED: 'seem<>SUBJ'];
```

7.4 Shared functional structures

As XLFG has been designed to share the analysis of ambiguous sentences, multiple F-structures are represented in a unique structure by using distributed features.

Distributed features

The ambiguous analysis for the sentence *John sees the man with a telescope* corresponds to these two F-structures:

```
[PRED: 'SEE<SUBJ, (OBJ)>',
 SUBJ: [PRED:'JOHN'],
 OBJ: [PRED:'MAN'],
 MOD: {[PRED: 'TELESCOPE' ]}]

[PRED: 'SEE<SUBJ, (OBJ)>',
 SUBJ: [PRED:'JOHN'],
 OBJ: [PRED:'MAN', MOD: {[PRED: 'TELESCOPE' ]}]
```

An economic way to represent this pair of F-structures is to share the equivalent attributes into a common factor and distribute the differences into a list written with vertical bar characters like this:

```
[PRED: 'SEE<SUBJ, (OBJ)>',
 SUBJ: [PRED:'JOHN'],
 OBJ: [PRED:'MAN'],

|
[ MOD: {[PRED: 'TELESCOPE' ]}]
,
[ OBJ: [MOD: {[PRED: 'TELESCOPE' ]}] ]
|

]
```

The `OBJ` attribute in the second F-structure is the result of unification between [OBJ: [PRED:'MAN']] and [OBJ: [MOD: {[PRED: 'TELESCOPE']}]].

7.5 Constraints on functional structures

A sentence is considered grammatical if the grammar can assign it with at least one constituent structure, and the Φ projection of that c-structure is a *coherent*, *complete*, *extended coherent* and *consistent* f-structure. These criteria result from implicit constraints of the theory, to which parochial constraints associated to lexical items or c-structure rules can be added.

Implicit constraints

Coherence

A functional structure is coherent if the attributes of all the governable functions it includes are specified on the PRED value, and all embedded f-structures are coherent.

Here is an example of an incoherent f-structure:
**Mary sleeps her bed.*

$$\begin{bmatrix} \text{PRED} & \text{'sleep<SUBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \\ \dots & \end{bmatrix} \\ \text{OBJ} & \begin{bmatrix} \text{PRED} & \text{'bed'} \\ \dots & \end{bmatrix} \end{bmatrix}$$

Completeness

An f-structure is complete if all the attributes specified in its PRED value occur and are instantiated locally, and if all embedded f-structures are complete.

Here is an example of an incomplete f-structure.
**Mary precedes.*

$$\begin{bmatrix} \text{PRED} & \text{'precede<SUBJ, OBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \\ \dots & \end{bmatrix} \end{bmatrix}$$

Extended Coherence

A functional structure is *extended* coherent if all the governable functions include a PRED value.

Here is an example of an *extended* incoherent f-structure:
**sleeps.*

$$\begin{bmatrix} \text{PRED} & \text{'sleep<SUBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \textit{PERSON} & 1|2|4|5|6 \end{bmatrix} \end{bmatrix}$$

Consistency

An f-structure is consistent if each local attribute has a unique value, and each embedded f-structure is consistent.

Here is an example of an inconsistent structure.
**The children sleeps.*

$$\begin{bmatrix} \text{PRED} & \text{'sleep<SUBJ>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'child'} \\ \text{NUMBER} & \text{sing} \mid \text{plur} \\ \ldots & \end{bmatrix} \end{bmatrix}$$

Explicit constraints

In addition to these implicit constraints, explicit constraints can be added through the use of equality, or difference constraints for local or non effects. In the following chapter we will describe such constraints and how functional structures are made.

7.6 Functional descriptions

Functional descriptions are a crucial part of LFG theory. They define the Φ projection allowing one to construct a functional structure from a constituent structure. They also provide explicit constraints on the output functional structures.

Syntactic relations, local and nonlocal agreement, subject control, subcategorization, etc., can be modelled using functional descriptions. The following examples are only illustrative, and the interested reader is directed to LFG literature for further reading and analyses.

Grammatical Functions

A list of terms which denote argument functions may be defined by the user. It allows them to distinguish them from the other grammatical functions. The argument functions specified in the `PRED` feature must be present in the local F-structure and all argument functions must be selected by their local `PRED`. These conditions correspond to traditional LFG contraints: Completeness Condition, Coherence Condition. In addition, the Extended Coherence Condition stipulates that all F-structures containing at least one argument function will also contain a `PRED`. The latter requirement is exposed in [Bresnan & Mchombo, 1987].

The keyword `functions` is used to define arguments functions:

```
functions: SUBJ, OBJ, XCOMP;
```

Without a declaration of argument functions, the completeness and coherence conditions are not respected.

Functional equations

Since "=" is implemented in XLFG as an assignment(see above chapt. 5.1), not equality; the only possibility of writing functional equations are the following.

↑ = ↓i; (where optional i is a number)
↓i = ↑;
(↑ <path>) = ↓i;
↓i = (↑ <path>);
(↑ <path>) = (↓i <path>);
(↓i <path>) = (↑ <path>);
(↑ <path>) = <atom>; (where atom stands for a functional structure, a string, an atomic value, etc.)
<atom> = (↑ <path>);

In these equations, one side affects the F-structure which is assigned, the other one affects a constant or a dominated F-structure.

Functional equations are needed to construct f-structures associated with c-structures. One may for instance assume a rule such as the following to associate a preverbal NP with the subject function:

```
S → NP VP
{
   (↑ SUBJ) = ↓1 ;
   ↑ = ↓2 ;
};
```

↓1 and ↓2 denote the f-structure of the dominated constituent, in the present instance respectively the `NP` and the `VP`, whereas ↑ denotes the functional structure of the dominating category, here `S`, the full sentence. When a rule introduces a single constituent, ↓ can be used equivalently to ↓1. The equation `(↑ SUJ) =` ↓1 instantiates a new attribute SUBJ in the f-structure ↑ and assigns the structure denoted by ↓1 as its value. If the attribute already existed in ↑, its value would become the unification of (↑ SUJ) and ↓1.

Let us take an example: the sentence *My father came.*

The functional structure initially assigned to the VP is:

$$\begin{bmatrix} \text{PRED} & \text{'come} < \text{SUBJ} > \text{'} \\ \text{TENSE} & \text{past} \\ \text{SUBJ} & \begin{bmatrix} \text{NUMBER} & \text{singular} \\ \text{PERSON} & 3 \end{bmatrix} \end{bmatrix}$$

The functional structure initially assigned to the `NP` is:

$$\begin{bmatrix} \text{PRED} & \text{'father'} \\ \text{NUMBER} & \text{singular} \\ \text{GENDER} & \text{masculine} \end{bmatrix}$$

Applying the equation (↑ SUJ) = ↓1 updates ↑ to:

$$\begin{bmatrix} \text{PRED} & \text{'come} < \text{SUBJ} > \text{'} \\ \text{TENSE} & \text{past} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'father'} \\ \text{NUMBER} & \text{singular} \\ \text{PERSON} & 3 \\ \text{GENDER} & \text{masculine} \end{bmatrix} \end{bmatrix}$$

We see here that the value of SUBJ is the unification of

$$\begin{bmatrix} \text{PRED} & \text{'father'} \\ \text{NUMBER} & \text{singular} \\ \text{GENDER} & \text{masculine} \end{bmatrix}$$

and

$$\begin{bmatrix} \text{NUMBER} & \text{singular} \\ \text{PERSON} & 3 \end{bmatrix}$$

Set-valued attributes

Attributes corresponding to modifiers have a set of f-structures as their value. The value of such attributes is constructed using declarations of the form

$\downarrow$i $\in$ ($\uparrow$ <path>)
($\downarrow$i <path>) $\in$ ($\uparrow$ <path>)

where ($\downarrow$i <path>) (or just $\downarrow$i) is the description of an f-structure X and ($\uparrow$ <path>) an attribute. If the attribute is already present in the structure as a set, X is added to it. If the attribute is present with another type, an error is reported.

As an example, let us consider a noun modified by an adjective and a relative clause: *the technical issues that plague the project.*

The following rules allow for an adequate analysis:

```
NP → DET N {
   ↑ = ↓1 ;
   ↑ = ↓2 ;
};

N → ADJ N
{
   ↓1 ∈ (↑ ADJ);
   ↑ = ↓2 ;
};

N → N REL
{
   ↑ = ↓1 ;
   ↓2 ∈ (↑ ADJ);
};
```

Here is the resulting f-structure, assuming a simplified lexicon:

$$\left[\begin{array}{ll} \text{PRED} & \text{'issue'} \\ \text{ADJ} & \left\{\begin{array}{l} \left[\begin{array}{ll} \text{PRED} & \text{'technical'} \\ \ldots & \end{array}\right] \\ \left[\begin{array}{ll} \text{PRED} & \text{'plague<SUBJ, OBJ>'} \\ \text{SUJ} & \left[\begin{array}{ll} \text{PRED} & \text{'PRO'} \\ \ldots & \end{array}\right] \\ \text{OBJ} & \left[\begin{array}{ll} \text{PRED} & \text{'project'} \\ \ldots & \end{array}\right] \\ \ldots & \end{array}\right] \end{array}\right\} \\ \ldots & \end{array}\right]$$

Links between f-structures

The use of equality in LFG functional equations allows for a single f-structure to be the common value of two attributes modelling distinct syntactic functions. This is how subject control is modelled in LFG: the subject of an `XCOMP` or `XADJ` is shared with a function of the matrix sentence. For instance the sentence *Mary wants to stay* is analyzed as:

$$\left[\begin{array}{ll} \text{PRED} & \text{'want<SUBJ, XCOMP>'} \\ \text{SUBJ} & \left[\begin{array}{ll} \text{PRED} & \boxed{1}\text{'Mary'} \\ \ldots & \end{array}\right] \\ \text{XCOMP} & \left[\begin{array}{ll} \text{PRED} & \text{'stay<SUBJ>'} \\ \text{SUBJ} & \boxed{1} \\ \ldots & \end{array}\right] \end{array}\right]$$

In this example, the subject of the infinitive is controlled by the matrix verb *want*, whose lexical entry states that it is shared with the subject of *want*—but the name *Mary* is realized just once in the constituent structure. In LFG, this effect is modelled by putting the following equation in the lexical entry of *want*:

$$(\uparrow \text{XCOMP SUBJ}) = (\uparrow \text{SUBJ})$$

Remember however that there is no equality in XLFG: equality is replaced by *assignment after unification*, a form of copy. But copy is not appropriate in the present instance: we do not want to state that the two subjects are identical, but that they are shared.

In XLFG, subject control is modelled using the operator "⇒":

$$(\uparrow \text{XCOMP SUBJ}) \Rightarrow (\uparrow \text{SUBJ})$$

This states that the left-hand side subject refers to the second one.

The general form of a link declaration is the following:

```
(↑ <path>) ⇒ (↑ <path>)
```

If the attribute corresponding to the left-hand side is already present with a feature-structure value, it must subsume the second one to produce a well-formed F-structure. If it is present, but with another type, an error is reported.

Conditionals

Since functional descriptions are assigned to phrase structure rules rather than constituents, we added the operators if that allows one to turn on or off the functional descriptions associated with optional constituents. Here is an example:

```
VP → AUX [advneg] VP
{
   if ($2)
      (↑ neg) = true;
   else
      (↑ neg) = false;
}
```

In this example, the f-structure of the VP will always carry a feature `neg`, with value `true` if a negative adverb is present, `false` otherwise.

Using the operator if is not needed if the functional description includes a reference to the functional structure of the optional term. The rule

```
NP → [DET] N
{
   ↑ = ↓1;
   ↑ = ↓2;
}
```

is equivalent to (and slightly awkward)

```
NP → [DET] N
{
   if ($1) ↑ = ↓1;
   ↑ = ↓2;
}
```

The general form for a conditional functional description is the following:

```
if ($i) <statement>
if ($i) <statement> else <statement>
if (not $i) <statement>
if (not $i) <statement> else <statement>
```

Selection

A compact and economical way to write LFG rules is to use the selection like this example from [Falk, 2001] p.76

$$\text{VP} \rightarrow \text{V} \quad \left\{ \begin{array}{c} \text{DP} \\ \text{NP} \end{array} \right\}^* \quad \text{PP}^* \quad \left(\left\{ \begin{array}{c} \text{CP} \\ \text{IP} \\ \text{S} \end{array} \right\} \right)$$

$$\uparrow = \downarrow \quad \left\{ \begin{array}{l} (\uparrow \text{OBJ}) = \downarrow; \\ (\uparrow \text{OBJ2}) = \downarrow; \end{array} \right\} \quad (\uparrow (\downarrow \text{PCASE})) = \downarrow \quad (\uparrow \text{COMP}) = \downarrow$$

The equivalent XLFG notation for term selection, is written by using "|":

```
VP → V [NPs] [PPs] [CP|IP|S];
```

In order to associate a function description with each selection, the keyword switch is used.

Here is the complete exemple:

```
VP → V [NPs] [PPs] [CP|IP|S]
{
   ↑ = ↓1;
   ↑ = ↓2;
   ↑ = ↓3;
   (↑ COMP) = ↓4;
};

NPs → [NPs] DP|NP;
{
   ↑ = ↓1;
   switch ($2) {
      case DP: (↑ OBJ) = ↓2;
      case NP: (↑ OBJ2) = ↓2;
}};

PPs → [PPs] PP;
{
   ↑ = ↓1;
   (↑ (↓ PCASE)) = ↓2;
};
```

The operator `switch` concerns `DP`|`NP` designated by $2: depending on whether this term is `DP` or `NP`, the different statements identified by the keyword case is applied.

The general form for a selection is the following:

```
switch ($i) {
   case <identifier>: <statement>
   case <identifier>: <statement>
   ...
}
```

Constraining equations

This is the analogue of LFG constraining equations, noted with the operator $=_c$. Such constraints do not build structure, but check that some attribute in a given f-structure has the required value.

As a possible application, notice that in English, finite clauses with the function of complement only optionally begin with a complementizer, whereas

finite clauses with the function of a subject need a complementizer. To account for this, we may assume that the complementizer *that* introduces a feature [CPLZER THAT], and that the c-structure rule for clausal subjects checks for the presence of that feature through an equality constraint:

```
S → S VP
{
   (↑ SUBJ) = ↓1 ;
   ↑ = ↓2;
   (↑ SUBJ CPLZER) == that;
}
```

As only an existing constant may be checked without building a structure, all the functional descriptions are accepted as equality operands:

<constant> == <constant>
Where <constant> is:
(↑ <path>)
(↓ <path>)
<atom>
<string>

Obviously, a constraint equation that makes reference to a constant which does not exist fails.

Negative constraints

The operator ≠ is the opposite of ==. A constraint such as the following is verified if either there is no CPLZER attribute in the structure, or its value is not THAT.

```
(↑ CPLZER) ≠ that;
```

<constant>) ≠ <constant>

This constraint fails if and only if the constants exist with values that do not match.

Variable attributes

A single verb may combine with two oblique complements. In such cases, the LFG practice is to index the syntactic function of the complement with the name of the adposition introducing it. This allows for a unique function to be assigned to each complement, in accordance with the unicity requirement on syntactic functions. Thus the PRED value for a verb such as *talk* is:

PRED: 'TALK<SUBJ, OBL_{to}, $\text{OBL}_{\text{about}}$ >'

To make sure that the right preposition is used within each complement, it is necessary to constrain the PCASE value associated with the preposition to match the indexed function, as in the following example.

$$\begin{bmatrix} \text{PRED} & \text{'TALK<SUBJ, OBL}_{\text{to}}\text{, OBL}_{\text{about}}\text{ >''} \\ \text{SUBJ} & [\ \dots\ ] \\ \text{OBL}_{\text{to}} & \begin{bmatrix} \text{PCASE} & \text{to} \\ \dots & \end{bmatrix} \\ \text{OBL}_{\text{about}} & \begin{bmatrix} \text{PCASE} & \text{about} \\ \dots & \end{bmatrix} \end{bmatrix}$$

In XLFG, such variable attribute names can be denoted by concatenating a description to the left of an attribute name: OBL - (f PCASE) names an attribute constructed by concatenating `OBL` with the value of the PCASE attribute of f using the operator "-". The following rule allows one to construct the preceding schematic f-structure from appropriate lexical entries.

```
VP → V [PP]
{
   ↑ = ↓1;
   (↑ OBL - (↓2 PCASE)) = ↓2;
}
```

Long distance dependencies

Long distance dependencies are standardly modelled in LFG through functional uncertainty, that is, the use of regular expressions in attribute path descriptions. This is readily implemented in XLFG. For instance, the following is a standard

rule for describing *wh-* questions in English such as *Who do you think John saw?*

```
S1 → NP S
{
   ↑ = ↓2;
   (↑ FOCUS) = ↓1;
   (↓ WH) == true;
   with $x in (↑ (COMP | VCOMP)*)
      ($x OBJ) ⇒ (↑ FOCUS);
}
```

In this description `(↑ (COMP | VCOMP)*)` denotes a sequence of COMPs and VCOMPs embedded in each other ↑. All values that correspond to this existing sequence are assigned to the variable `$x`. `($x OBJ)` denotes the OBJs embedded in them.

The general structure of such statements is

```
with $<identifier> in (↑ <regexp>)
   <statement>
```

where `X` names the function of the fronted constituent in the embedded clause, and <`regexp`> is a regular expression over the set of attribute names.

A regular expression denotes a path in a functional structure. The simplest kind of regular expression is just an attribute name. From two regular expressions `A` and `B`, one can derive the complex expressions `(A B)` `(A|B)` and `A*`, corresponding respectively to concatenation, disjunction, and iterative closure.

Let us take a few examples from the f-structure of *John asks which file Luke should open.*

$$\begin{bmatrix} \text{PRED} & \text{'ask<SUBJ, (COMP)>'} \\ \text{SUBJ} & \boxed{x}\begin{bmatrix} \text{PRED} & \text{'John'} \end{bmatrix} \\ \boxed{l} & \\ \text{COMP} & \boxed{y}\begin{bmatrix} \text{PRED} & \text{'should} < \text{SUBJ, VCOMP} >' \\ \text{FOCUS} & \boxed{k}\begin{bmatrix} \text{PRED} & \text{'file'} \end{bmatrix} \\ \text{SUBJ} & \boxed{z}\begin{bmatrix} \text{PRED} & \text{'Luke'} \end{bmatrix} \\ \text{VCOMP} & \boxed{w}\begin{bmatrix} \text{PRED} & \text{'open} < \text{SUBJ, OBJ} >' \\ \text{SUBJ} & \Rightarrow \boxed{z} \\ \text{OBJ} & \Rightarrow \boxed{k} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

(↑ (COMP SUBJ)) denotes structure [PRED 'Luke']—that is, z.

(↑ (COMP | SUBJ)) denotes the structures [PRED 'should'] or [PRED 'John']—that is, neither exclusive to x or y.

(↑ (COMP | VCOMP)∗) denotes all the structures l, y, w.

Complex predicates

A feature of LFG is that it is impossible to unify two structures with distinct PRED features. This is the standard way of ensuring that each syntactic function is instantiated no more than once by PRED-bearing constituents, without barring the possibility that distinct constituents contribute to the description of an f-structure.

This feature of LFG can be put to use, for instance, to ensure that controlled infinitives will not get a c-structure subject. The f-structure of *Mary promised to come* is:

$$\begin{bmatrix} \text{PRED} & \text{'promise<SUBJ, XCOMP>'} \\ \text{SUBJ} & \boxed{1}\begin{bmatrix} \text{PRED} & \text{'Mary'} \\ \ldots & \end{bmatrix} \\ \text{XCOMP} & \begin{bmatrix} \text{PRED} & \text{'come} < \text{SUBJ} >' \\ \text{SUBJ} & \Rightarrow \boxed{1} \end{bmatrix} \end{bmatrix}$$

Notice that the matrix and embedded SUBJ are identified, thanks to a control specification originating in the lexical entry of *promise*:

$$(\uparrow \text{XCOMP SUJ}) \Rightarrow (\uparrow \text{SUJ})$$

By contrast, the f-structure of *Mary promised that she would come* is:

$$\begin{bmatrix} \text{PRED} & \text{'promise<SUBJ, COMP>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'Mary'} \\ \dots & \end{bmatrix} \\ \text{COMP} & \begin{bmatrix} \text{PRED} & \text{'come} < \text{SUBJ} > \text{'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'PRO'} \\ \dots & \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Here the two subjects have distinct f-structures corresponding to distinct PRED values (the fact that they might be coindexed semantically is a separate issue that we do not model here).

Now let us consider what the f-structure of a sentence such as **Mary$_i$ promised [her$_i$ to come]* would be. Assuming that a well-formed c-structure could be assigned to this sentence, its f-structure would be:

$$\begin{bmatrix} \text{PRED} & \text{'promise<SUBJ, XCOMP>'} \\ \text{SUBJ} & \boxed{1}\begin{bmatrix} \text{PRED} & \text{'Mary'} \\ \dots & \end{bmatrix} \\ \text{XCOMP} & \begin{bmatrix} \text{PRED} & \text{'come} < \text{SUBJ} > \text{'} \\ \text{SUBJ} & \boxed{1} \mid \begin{bmatrix} \text{PRED} & \text{'PRO'} \\ \dots & \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

This f-structure is ill-formed, because the PRED values 'PRO' cannot subsume — a fortiori unify — 'Mary', despite the fact that the f-structures they occur in are constrained to being identified by the control equation.

This said, it is well known that *complex predicate constructions* rest on a situation where two distinct constituents contribute to the specification of a PRED value. Particle verbs, support verb constructions, decomposable idioms, and serial verb constructions are examples of cases that may be modelled as complex predicates.

To model such cases, XLFG supports an operator prefix or suffix "-" that derives a PRED value from two other PRED values. The predicate name is the concatenation of the two predicate names, and the set of functions is the union of the sets of functions of the two PRED-bearing constituents.

PRED: 'lexeme <...> ...'
PRED: 'prefix - <...> ...'
PRED: ' - suffix <...> ...'

Let us illustrate this situation with a support verb construction: *give a lecture*. This is a partially grammaticalized construction: both *give* and *lecture* seem to have their usual meaning, but (i) something must be said to the effect that *give* is used rather than other candidate verbs such as *make* or *do*, and (ii) the verb seems to inherit something from a valence requirement originating in

the noun: in the following sentence, *on the subject* is a complement of the verb, but it is the noun *lecture* and not the verb *give* that is lexically specified for an oblique complement in *on*[1].

The lecture he gave on the subject in Salzburg was judged as one of the turning points in the evolution of theoretical physics. (A. Calaprice & T. Lipscombe, *Albert Einstein: a biography*, p. 46, Greenwood Publishing Group 2005)

The lexical entries are as follow.

give V [PRED: 'give - <SUBJ>', tense: present];
lecture N [PRED:'lecture<(onOBL)>', number: singular];

The two structures may unify to produce an appropriate f-structure for the sentence above:

$$\begin{bmatrix} \text{PRED} & \text{'give-lecture<SUBJ, (onOBL)>'} \\ \text{SUBJ} & \begin{bmatrix} \text{PRED} & \text{'PRO'} \\ \dots & \end{bmatrix} \\ \text{onOBL} & \begin{bmatrix} \text{PRED} & \text{'SUBJECT'} \end{bmatrix} \\ \dots & \end{bmatrix}$$

Here, in summary, are the various combinations for unification between PRED in XLFG:

	Prefix	Suffix	Lexeme
Prefix	Prefix	None	Lexeme
Suffix	None	Suffix	Lexeme
Lexeme	Lexeme	Lexeme	None

As one can see, it is always possible to unify more than two PRED, thanks to the idempotence property of unification on prefix and suffix. This possibility may happily be used for serial verb analysis.

7.7 XLFG lexica

Although XLFG has been developed to extract syntactic properties from sentences, but not for phonological or morpho-syntactical treatment, it allows us to carry out a basic analysis of compounds or portmanteau forms.

Words are written in UTF-8 encoding. The system will accept some accented characters directly, but when using non latin alphabets or less used symbols or keywords, double quotes should be added.

In order to parse numbers (which correspond to an infinite regular string langage) without using the power of the XLFG parser and with a finite lexica,

[1] Thanks to Olivier Bonami for his remarks.

we added special forms: `_INTEGER_` and `_REAL_`. The first matches a digital string, followed by a real number written in different scripts.

Here some exemples of accepted forms:

```
John
"emergency exit"
"日本語"
Schreibmaschinenpapier
_INTEGER_
";"
```

An XLFG lexical entry consists of a triplet (category label, functional structure, set of local functional constraints). Functional structures and functional constraints are optional.

A simple form is associated with one triplet, while an homonym form is associated with several ones separated with a vertical bar character.

Here an example of homonym entries for the form *left*: past tense of *leave* or opposite of *right*.

```
left commonNoun [PRED: 'LEFT'] | verb [PRED:'LEAVE'];
```

A poly-categorial word (compound or portmanteau word that must be analysed according to a morphological theory, or agglutinate word) is represented by a list of triplets separated with the ampersand character &. For example, the French word *auquel* is the agglutination of the preposition *à* and the relative pronoun *lequel*

```
auquel (prep [PRED: 'à']
        & relPro [PRED: 'lequel', GENDER: ms, NUMBER: sg]);
```

Obviously, one may combine these two possibilities. For example, the little French word *du* is either a partitive determiner, or a definite article *le* following the preposition *de*.

```
du det [NUMBER: sg, PARTITIVE: true, DEFINED: false]
   | (prep [PRED: 'DE', PCASE: DE]
      & det [GENDER: ms, NUMBER: sg, DEFINED: true]);
```

The F-structure in an lexicon entry may be followed by local functional constraints. It allows us to give the syntactic property of the word depending on its context.

Let us take the example of a subject control verb such as *want*. An optimal lexical entry will look like this:

```
wants v [PRED:'WANT<SUBJ.agent, VCOMP.theme>',
         TENSE: present, SUBJ: [NUMBER: sg, PERSON: 3]]
         {
            (↑ VCOMP SUBJ) ⇒ (↑ SUBJ);
         };
```

Information on the nature of the predicate and subject agreement are constant in uses of this entry, so they should be specified in the F-structure. Yet the constraint linking the subject of the infinitive to the local subject depends on the context and should thus be stated separately. Thus the only context anchor for the functional constraints in reference to a lexical entry is ↑, not ↓ *i*.

7.8 Macros

As a lot of similar attributes with the same values are used, we added a convenient way of writing this only once using an assigned variable marked with "@":

```
@ms: GENDER: masc, NUMBER: sing;
```

It is possible to use this variable anywhere in the lexicon, in the grammar, or in macro definition:

```
@ms: GENDER: masc, NUMBER: sing;
@K: VFORM:participle;
@Kms: @K, @ms;
```

7.9 Unknown words

When a unknown word is encountered, XLFG assigns it withthe special value `_UNKNOWN_`. One can associate open categories (nouns, verbs, adjectives,) with the unknown words, but not grammatical lexemes such as preposition, particules or determiners.

In this case, and also in the case of regular expressions which we have already discussed, the keyword `_TEXT_` corresponds to the exact form encountered in the input. This enables us to rewrite this form in the calculated F-structure.

```
_UNKNOWN_ verb[PRED: '_TEXT_']
          | noun[PRED: '_TEXT_']
          | adjective[PRED: '_TEXT_']
          | adverb[PRED: '_TEXT_'];
```

7.10 Compiled lexicon

The back-end of XLFG has been designed to work with large lexica. Thus a compiled lexicon can be given to the parser in addition to the explicit lexicon. In the case of conflitbewteen the two, the explicit lexicon overrides the other.

However, the front-end of XLFG does not enable this feature because it has been designed to experiment the parser in other ways with real applications through Natural Language Processing that XLFG back-end can deal with.

Furthermore, we provide an onsite consulting service which is incompatible with a huge lexicon.

For further information on the software's capabilities, please contact the author.

8 APPENDIX

8.1 XLFG keywords

```
PRED
LEXEME
FALSE
TRUE
_INTEGER_
_UNKNOWN_
_REAL_
_TEXT_
case
else
functions
if
in
not
startSymbol
switch
with
```

8.2 Terms

↑ The F-structure corresponding to the node where this term is noted.

↓**n** The F-structure associated with the nth node from left to right. It is noted that an empty node is counted with the others. ↓ denotes ↓1

<**atom**> An atomic value is a list of identifiers or integers separated by vertical bar characters. The result is one of the listen values.

<**string**> A string is noted with double quotes characters. Any UTF-8 character may be included in a string. In order to design the double quote itself, one may use the escape character "\".

<**symbol**> Symbols are noted with alpha-numeric latin characters beginning with an alpha. In order to use non latin characters, one may use the character " ‘ ". Example:

8.3 Grammar description syntax for statements

<statement> ::= { <list_statement> }
| <up_path> = <atom> ;
| <atom> = <up_path> ;
| <up_path> = <string> ;
| <string> = <up_path> ;
| <path> == <atom> ;
| <atom> == <path> ;
| <path> == <string> ;
| <string> == <path> ;
| <path> ≠ <atom> ;
| <atom> ≠ <path> ;
| <path> ≠ <string> ;
| <string> ≠ <path> ;
| <up_path> ⇒ <up_path> ;
| <up> = <down> ;
| <down> = <up> ;
| <up_path> = <down> ;
| <down> = <up_path> ;
| <up_path> = <down_path> ;
| <down_path> = <up_path> ;
| <up_path> = <feature_structure> ;
| <feature_structure> = <up_path> ;
| <down> ∈ <up_path> ;
| <down_path> ∈ <up_path> ;
| if (<dollar>) <statement>
| if (<dollar>) <statement> else <statement>
| if (not <dollar>) <statement>
| if (not <dollar>) <statement> else <statement>
| switch (<dollar>) { <list_case_statement> }
| with <variable> in (<up> <regExpE>) <statement>

<list_case_statement ::= <case_statement> <list_case_statement>
| <case_statement>

<case_statement> ::= case <stringOrIdentifier> : <statement>

<path> ::= (<up> <path_cdr>)
| (<down> <path_cdr)>

<path_cdr> ::= <path_cdr> <step_path>
| <step_path>

<step_path> ::= <atomUniqStr>
| <down_path>

| <step_path> - <step_path>

<up> ::= ↑
| <variable>

<variable> ::= $ <identifier>

<down> ::= ↓
| ↓ <integer>

<dollar> ::= $ <integer>

<regExpE> ::= <regExpE> <regExpT>
| <regExpT>

<regExpT> ::= <regExpT> "|" <regExpF>
| <regExpF>

<regExpF> ::= (<regExpE>)
| <regExpF> *
| <regExpF> +
| <regExpF> ?
| <atomUniq>

<atom> ::= <atomUniq>
| <atomUniq> "|" <atom>

<atomUniq>::= <identifier>
| <integer>

References

[Bresnan, 1995] Bresnan, Joan. 1995. Linear Order, Syntactic Rank, and Empty Categories: On Weak Crossover. *Pages 241–274 of:* Dalrymple, Mary, Kaplan, Ronald M., Maxwell, John T., & Zaenen, Annie (eds), *Formal Issues in Lexical-Functional Grammar*. Stanford, CA: CSLI Publications.

[Bresnan, 2001] Bresnan, Joan. 2001. *Lexical-Functional Syntax*. Oxford: Blackwell Publishers.

[Bresnan & Mchombo, 1987] Bresnan, Joan, & Mchombo, Sam A. 1987. Topic, pronoun, and agreement in Chichewa. *Language*, **63**(4), 741–782. Reprinted in Masayo Iida, Steven Wechsler, and Draga Zec (eds.), Working Papers in Grammatical Theory and Discourse Structure: Interactions of Morphology, Syntax, and Discourse, pp. 1–59. Stanford: CSLI Publications.

[Butt *et al.* , 1999] Butt, Miriam, Niño, María-Eugenia, & Segond, Frédérique. 1999. *A Grammar Writer's Cookbook*. Stanford, CA: CSLI Publications.

[Dalrymple, 2001] Dalrymple, Mary. 2001. *Lexical Functional Grammar*. Syntax and Semantics, vol. 34. New York: Academic Press.

[Falk, 2001] Falk, Yehuda N. 2001. *Lexical-Functional Grammar: An Introduction to Parallel Constraint-Based Syntax*. Stanford, CA: CSLI Publications.