HAL open science

XLFG Documentation

Lionel Clément

To cite this version:

Lionel Clément. XLFG Documentation. 2016. hal-01277648v2

HAL Id: hal-01277648

https://hal.science/hal-01277648v2

Preprint submitted on 3 Mar 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

XLFG Documentation

version 9.8.0 – February 2016

Lionel Clément

Contents

1 What is XLFG? 4

2 Installing XLFG 5

3 Accounts 6

4 First steps with XLFG 7
4.1 Using one of the sample grammars . . . 7
4.2 Changing the example . . . 9
4.3 Error messages . . . 9
4.4 Modifying the grammar . . . 9

5 LFG theory and XLFG parsing 11
5.1 Solving equations . . . 11

6 Using the XLFG Web-portal interface 13

7 XLFG grammar 16
7.1 XLFG Notations . . . 16
7.2 Phrase structure rules . . . 18
7.3 Functional structures . . . 19
7.3.1 Atomic or literal feature . . . 20
7.3.2 The feature `PRED` . . . 20
7.3.3 Complexe predicates . . . 22
7.3.4 The feature `LEXEME` . . . 24
7.3.5 The feature `SUBCAT` . . . 25
7.3.6 Lexical entries for Complexe Predicates . . . 25
7.3.7 Embedded feature-structures . . . 26
7.3.8 Sets of feature-structures . . . 27
7.4 Shared functional structures . . . 27
7.5 Constraints on functional structures . . . 28
7.6 Functional descriptions . . . 30
7.7 XLFG lexica . . . 41
7.7.1 Unknown words . . . 43

7.8 Macros 43
7.9 Compiled lexicon 44

8 Ranking **45**

9 APPENDIX **46**

Acknowledgment

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

Olivier Bonami (Univ. Paris-Sorbonne)

Kim Gerdes (Sorbonne Nouvelle, Paris)

Sekou Diao (Univ. Bordeaux)

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 9.8.0 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 hundredths 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 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.

3 Accounts

Checking in is not mandatory for trying 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 organizing 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 project including a grammar with `read-only` privilege and a lexicon with `read-write` privilege for our example.

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

The teacher may change from `read-write` to `read-only` mode the lexicon after a deadline, and read and correct each result.

Privilege	Enrolled in the Workshop	Workshop Owner
hidden	not visible	shared data modifiable
read-only	visible but not modifiable	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 *Grammars* menu. A page opens with a number of pre-filled fields.

Declaration This part will contain some useful declarations. It's not helpful for us in this first example.

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.

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

Parsing contains the input sequence that will be parsed: *John sees the man with a telescope* in our first example. It also 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).
- An argument structure for the sentence (fig. 5).

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

As this number can be quite large or unlimited, XLFG displays only one tree at a time or a shared representation, namely a forest-representation of the constituent-structures.

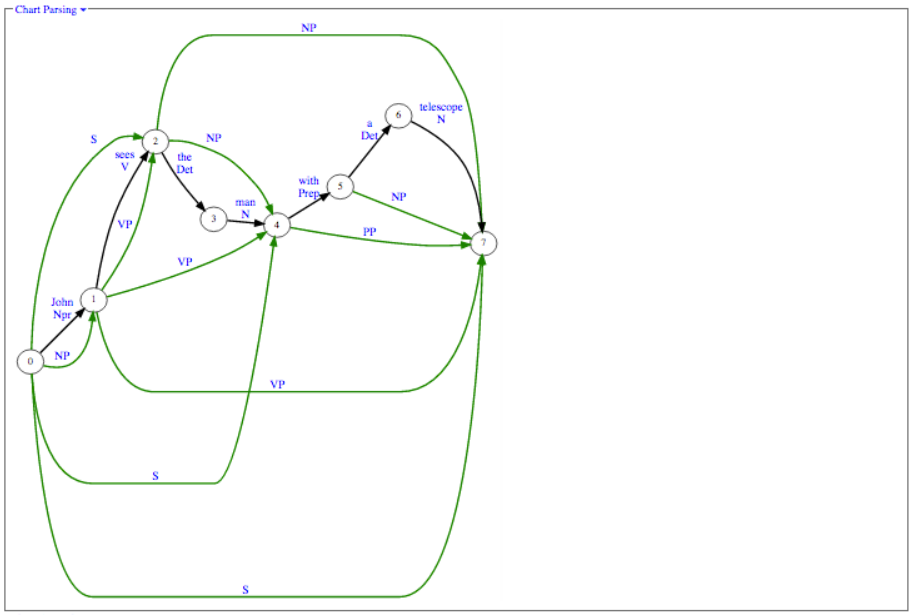

Figure 2: Lattice of words and phrases

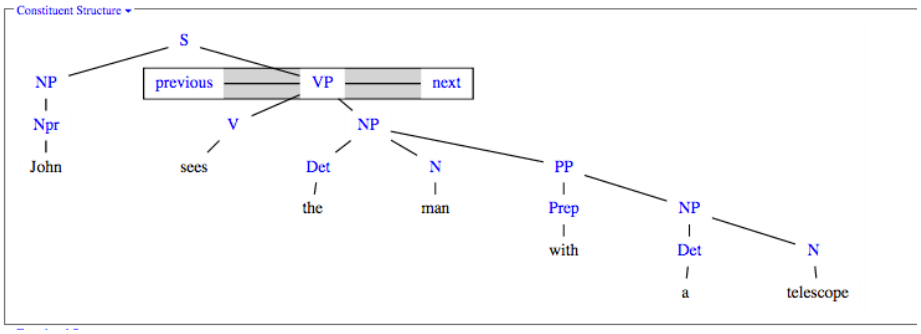

Figure 3: Constituent-Structure

In our exemple, to see the other constituent-structures and the related functional-structures, 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*).

Click on the other links of the graphes to see the related functional-structures and argument-structures.

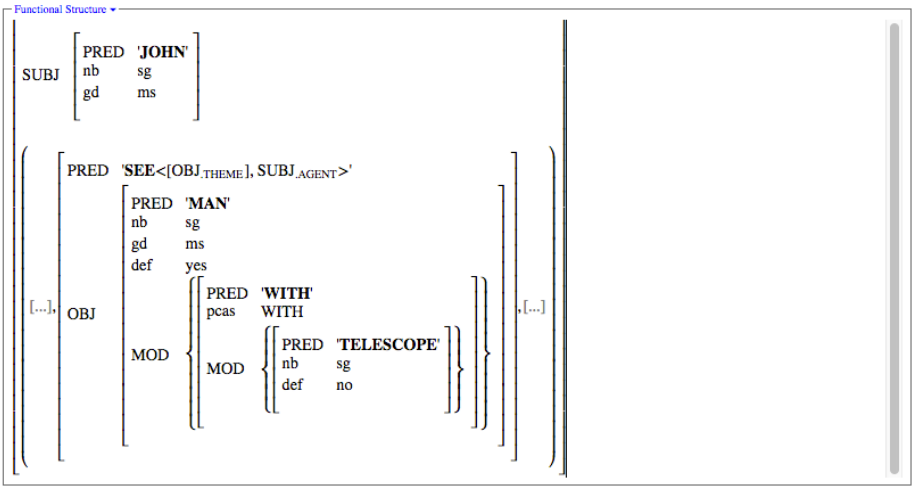

Figure 4: Functional-Structure

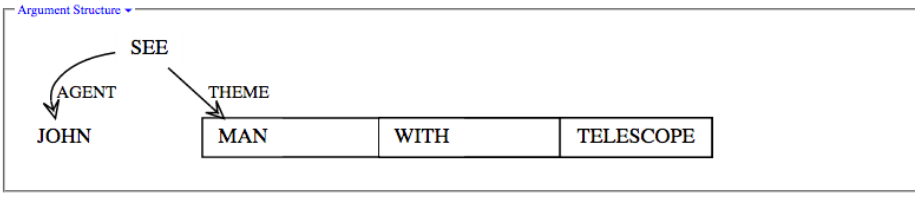

Figure 5: Argument-Structure

4.2 Changing the example

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 careful about this when writing a grammar.

4.3 Error messages

If you make a mistake, you will get a (hopefully helpful) error message which provides a line number and the project part (declaration, grammar, or lexicon).

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 how grammars found in LFG books can be implemented into XLFG.

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.

Furthermore, XLFG allow us to adapt the main principles agreed in the LFG literature in a simple and elegant manner.

Formal definition of = symbol in XLFG

Given a shared forest where the node 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 $\phi \neq \perp$, 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:

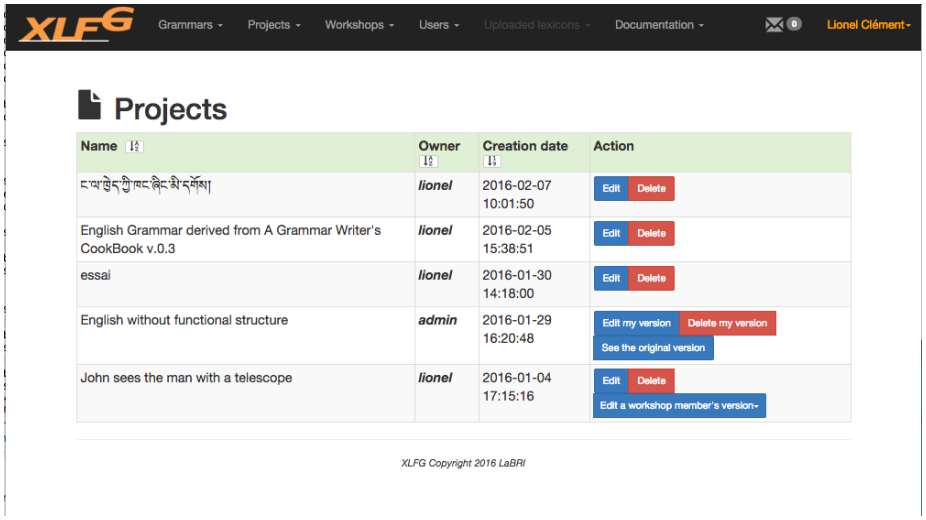

Figure 6: Home XLFG Web-portal

Grammars

The `Grammars` menu contains some XLFG grammars in further languages with the aim of facilitating XLFG development.

These freely grammars was developed by XLFG contributors and XLFG developer. Thanks a lot to all contributors who are helping us to improve XLFG data!

Projects

Each user may edit and save projects involving a grammar, a lexicon and output parameters.

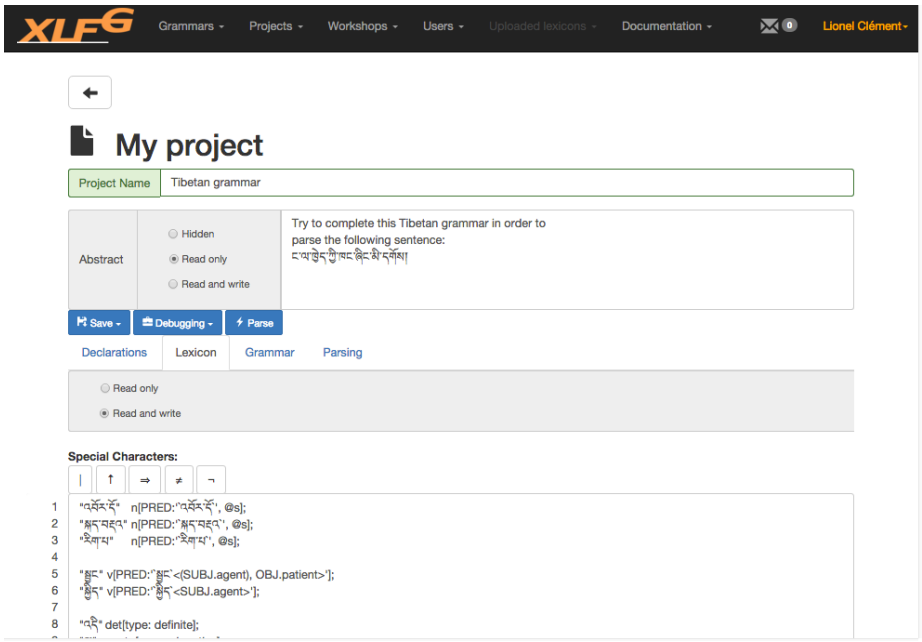

Figure 7: Project page

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

Any member enrolled in a workshop can copy and then edit a new version of a shared project. Only the `read-write` parts can be then edited, and the workshop owner may read and correct this version. It is very efficient for organize tutorial. Of course, in addition, any user can make a personal copy of any readable project. This personal copy is not readable by the workshop owner by default.

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.

Users

Any registered user is listed here with workshops and projects he or she manages. This interface allows anyone to send a notification to a user and to ask to join his workshops. We hide all emails. The communications between users can only

be with XLFG notifications.

Notifications

Any recorded user may send notifications to the workshop administrators, 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.

At the request of XLFG users, XLFG can automatically send an email to the users when a new notification arrives.

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.

Identifiers

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, θ, 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
- SUBCAT
- case
- else
- functions
- if
- in
- not
- start
- switch
- symbol
- with

Identifiers are used on different occasions:

- As a terminal symbol in the lexicon and grammatical rules (`common_noun`, `verb`, `preposition`, ...).
- As a non-terminal symbol in grammatical rules (`S`, `NP`, `PP`, ...).
- As an attribute or atomic value in feature structures (`number`, `singular`, ...).
- As a grammatical function (`SUBJECT`, `OBJECT`, ...).
- As argument (`Agent`, `Patient`, ...).

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

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

The maximal number of different identifiers in the same project is limited to 248, witch is suffisant for building large coverage XLFG grammars for many languages.

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 following LFG rule:

VP → 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 modeling 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 keywords `start symbol` in the declaration section of the project.

```
start symbol: 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
 SUBJ: [PRED: 'John', NUMBER: sg, person: 3]]
```

7.3.1 Atomic or literal feature

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

- An atomic value is a number, an identifier, the symbols `+` or `-`, or a list of atomic values separated by the symbol |.

```
TENSE: present,
PERSON: 1 | 2,
DEF: +,
MOOD: indicative | subjunctive
```

The number of atomic values is limited to 251.

- A literal value is a symbol marked between simple quotes.

```
PREPOSITIONAL_FORM: 'into',
```

The number of literals is not limited.

7.3.2 The feature `PRED`

While the attribute is `PRED`, the value is written between two single quotes. A regular lexeme is noted with a single symbol followed by sub-categorization information.

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. Optional grammatical functions are noted with [ ] brackets.

(1) a. The Commission would like to give some of the results that are included in the document.

b. The Commission seems to give to the shareholders some of the results.

```
PRED: 'TO_GIVE < SUBJ, OBJ, [OBLto] >'
PRED: 'TO_SEEM < XCOMP > SUBJ'
```

If a list of grammatical functions is allowed for the same syntactic position, the grammatical functions are separated with | symbol.

(2) a. The conclusions say that a new approach is needed.
b. The Commission is unable to say the degree of progress.

PRED: 'TO_SAY < SUBJ, [OBJ | SCOMP] >'

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 parser, but in an other software.

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

(3) a. The Commission wants to place the shareholder at the center of his commitment.

Here is a sample PRED feature in the XLFG notation:

PRED: 'TO_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`. In this case, the `OBL` locative complement is not mandatory.

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: 'TO_RAIN <> SUBJ'

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

PRED: TO_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.

(4) a. The computer crashed.
b. Luke crashed the computer.

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

PRED: 'TO_CRASH < SUBJ . patient >'
PRED: 'TO_CRASH < SUBJ . agent, OBJ . patient >'

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

7.3.3 Complexe 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.

Let consider these sentences:

(5) a. Mary promised to come.
b. Mary promised that she would come
c. *Mary$_i$ promised [her$_i$ to come]

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 5a 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 5b is:

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

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 the agrammatical sentence 5c. Assuming that a well-formed c-structure could be assigned to this sentence, its f-structure would be:

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

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 modeled 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:

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

Here, in summary, are the various combinations for unification between two PRED attributes 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.

7.3.4 The feature `LEXEME`

In order to describe only the lexeme attribute but not the subcategorization of a predicate, one may use the attribute `LEXEME` instead of `PRED`.

This attribute can combine with a `PRED` to form a *complex predicate* without changing anything about the lexical valence given by the main predicate.

The lexical entries for a verbal particule *look for* may be encoded as follow

[1]Thanks to Olivier Bonami for his remarks.

look V [PRED: 'TO_LOOK < SUBJ, [OBJ] >', tense: present];
for VERB_PART [LEXEME: ' - FOR '];

The unify result is the compositional predicate

[PRED: 'TO_LOOK_FOR < SUBJ, [OBJ] >', tense: present]

7.3.5 The feature `SUBCAT`

By contrast, a value for `SUBCAT` attribute is the subcategory part of a `PRED` attribute except the `lexeme`.

SUBCAT: '<Subj.agent>'

A `SUBCAT` feature may unify with a `PRED`. In such a case, the result is a `PRED`.

Here, in summary, are the various combinations for unification between `PRED`, `LEXEME` and `SUBCAT` attributes in XLFG:

	PRED:'X<Y>Z'	LEXEME:'X'	SUBCAT:'<Y>Z'
PRED:'A<B>C'	PRED:'X U A<B U Y>C U Z'	PRED:'X U A<B>C'	PRED:'A<Y>Z'
LEXEME: 'A'	PRED:'X U A<Y>Z'	LEXEME:'X U A'	PRED:'A<Y>Z'
SUBCAT: '<B>C'	PRED:'X<B>C'	PRED:'X U A<B>C'	none

7.3.6 Lexical entries for Complexe Predicates

In many cases where the predicate depends on a combination of two components, the meaning and the argument structure depend on a indecomposable idiom. That is the case for a particule verb for exemple.

The predicate of the particule verb *to give up* is `give_up` that don't be the combination between *to give something to someone* and the preposition *up something*. Moreover, the subcategorization if particule verb is different from the subcategorization of *to give*.

(6) a. He gave a toy to a child.
b. He gave me his phone number.

c. He gave a concert.

d. They gave up their personal possessions.

In order to introduce a new lexical entry from such a combination, we use the `#` symbol followed by the new lexeme as follow:

```
# TO_GIVE_UP [SUBCAT: '< SUBJ, [XCOMP | OBLon] >'];
```

Given the lexical entries for the main verb *to give* and the particule *up* encoded as follow:

```
give V [PRED: 'TO_GIVE<SUBJ, OBJ, [OBLto | OBL]>', tense: present];
up VERB_PART [LEXEME:' - _UP'];
```

The feature-structure that has resulted from unification is the following:

- the lexeme is the combination between `TO_GIVE` and `-_UP`,
- The subcategorization is given only by the # lexical entry,
- The others features are given by the unification of the third lexical entries (verb, particule, verbal particule)

```
[PRED: 'TO_GIVE_UP < SUBJ, [XCOMP | OBLon] >', tense: present];
```

7.3.7 Embedded feature-structures

Feature values for grammatical functions are embedded feature-structures

```
[PRED: 'TO_SNORE<SUBJ>',
tense: present,
SUBJ: [PRED: 'BOY'] ]
```

7.3.8 Sets of feature-structures

A set of F-structures is written within braces:

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

Such a set of F-structures is used to describe an unordered list of modifiers.

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']}]].

An other exemple is the unique lexical entry for a regular English verb without `s` ending. It's encode both a plural subject and singular first or second person subject.

A useful way to encode the two entries is the following:

```
eat verb [
PRED: 'TO_EAT<SUBJ, [OBJ]>',
SUBJ: [ ( [ number:singular, person:1|2], [ number:plural ] ) ]
];

eats verb [
PRED: 'TO_EAT<SUBJ, [OBJ]>',
SUBJ: [ number:singular, person:3]
];
```

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.*

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

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.*

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

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.*

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

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 several logical operators between local or non local features. 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, sub-categorization, etc., can be modeled 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 and Mchombo, 1987].

The keyword `functions` is used to define arguments functions in the declaration section of a project:

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

Without such a declaration of argument functions, the Completeness Condition, Coherence Condition, and Extended Coherence Condition are not verified.

Functional equations

Since the symbol `=` 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>) = <value>; (where value stands for a functional structure or an atomic value)
<value> = (↑ <path>);
(↑ PRED) = '<pred>'; (where pred stands for a predicate value)
'<pred>' = (↑ PRED)
(↑ LEXEME) = '<lexeme>'; (where lexeme stands for a lexeme value)
(↑ SUBCAT) = '<subcat>'; (where subcat stands for subcategorization data)
'<subcat>' = (↑ SUBCAT);

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 ;
};
```

$\downarrow$1 and $\downarrow$2 denote the f-structure of the dominated constituent, in the present instance respectively the `NP` and the `VP`, whereas $\uparrow$ denotes the functional structure of the dominating category, here `S`, the full sentence. When a rule introduces a single constituent, $\downarrow$ can be used equivalently to $\downarrow$1. The equation `(`$\uparrow$ `SUJ) =` $\downarrow$`1` instantiates a new attribute SUBJ in the f-structure $\uparrow$ and assigns the structure denoted by $\downarrow$1 as its value. If the attribute already existed in $\uparrow$, its value would become the unification of ($\uparrow$ SUJ) and $\downarrow$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 ($\uparrow$ SUJ) = $\downarrow$1 updates $\uparrow$ 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 (↓i <path>) (or just ↓i) is the description of an f-structure X and (↑ <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:

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

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 modeling 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:

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

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 modeled 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*. But copy is not appropriate in the present instance: we do not want to construct a new subject.

In XLFG, subject control is modeled using the operator $\Rightarrow$:

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

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

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

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

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 $\begin{bmatrix} \text{CPLZER} & \text{THAT} \end{bmatrix}$, 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>
<literal>

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 matches.

Existential constraint

It is possible for a syntactic rule or a lexical entry to require a feature to be present without a particular value. For exemple, a finite verb is required when the complementizer *that* is the head of a clause whatever is the tense of such a verb.

(7) a. The Commission is also still debating whether to apply the law.
b. *The Commission is also still debating if to apply the law.
c. The Commission is also still debating if one should apply the law.

We use only the functional description to describe such a constraint.

```
that CPLTZ [] {(↑ TENSE);};
```

Conversely, a syntactic rule or a lexical entry may require a feature to be not present. For exemple, a verb introduced by the `to` particule is no finite. We use the ¬ symbol to represent describe constraint.

(8) a. *The Commission is also still debating whether to should apply the law.

```
to PART [] {¬(↑ TENSE);};
```

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.

$$\begin{bmatrix} \text{PRED} & \text{'TALK<SUBJ, OBLto, OBLabout >'} \\ \text{SUBJ} & [\ \ldots\ ] \\ \text{OBLto} & \begin{bmatrix} \text{PCASE} & \text{to} \\ \ldots & \end{bmatrix} \\ \text{OBLabout} & \begin{bmatrix} \text{PCASE} & \text{about} \\ \ldots & \end{bmatrix} \end{bmatrix}$$

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 modeled 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.*

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

(↑ (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.

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 words witch correspond to a local grammar (i.e. recognizable with a regular expression) without using the power of the XLFG context-free parser and with a finite lexica, we added special forms: `_EMAIL_`, `_URL_`, `_INTEGER_`, and `_REAL_`. The first matches a regular email adresse as `lionel.clement@xlfg.org`, the second matches an url as `http://www.xlfg.org`, etc.

Here some exemples of accepted forms:

```
John
"emergency exit"
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 | symbol.

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 analyzed 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.

7.7.1 Unknown words

When a unknown word is encountered, XLFG assigns it with the 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 `_THIS_` corresponds to the form encountered in the input. This enables us to rewrite this form in the calculated F-structure.

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

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 in the definition section

```
@m: GENDER: masc;
@s: NUMBER: sing;
@p: NUMBER: plural;
@ms: @m, @s;
@K: VFORM:participle;
@Kms: @K, @ms; @_12: PERSON: 1|2;
@P: tense: present, mood: indicative;
@V12s_p: SUBJ: [ ( [@_12, @s], [@p] ) ];
@P12s_p: @P, @V12s_p;
```

Then every macros may be used in the grammar or in the lexicon:

give V [PRED: 'TO_GIVE', @P12s$_p$];

7.9 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 conflit between the two, the explicit lexicon overrides the other.

XLFG has granted the users registered as teachers or XLFG contributors to upload lexicons. Such a lexicon is a simple text file (i.e., no formatting) where each line represents one lexical form in XLFG format.

In order to use a large lexicon into a project, a teacher or an XLFG contributor may just attach the lexicon name withit.

8 Ranking

To appear

9 APPENDIX

XLFG keywords

```
PRED
LEXEME
SUBCAT
case
else
functions
if
in
not
start
switch
symbol
with
```

XLFG local grammars

```
_INTEGER_
_FLOAT_
_URL_
_EMAIL_
```

References

[Bresnan, 1995] Bresnan, J. (1995). Linear order, syntactic rank, and empty categories: On weak crossover. In Dalrymple, M., Kaplan, R. M., Maxwell, J. T., and Zaenen, A., editors, *Formal Issues in Lexical-Functional Grammar*, pages 241–274. CSLI Publications, Stanford, CA.

[Bresnan, 2001] Bresnan, J. (2001). *Lexical-Functional Syntax*. Blackwell Publishers, Oxford.

[Bresnan and Mchombo, 1987] Bresnan, J. and Mchombo, S. A. (1987). Topic, pronoun, and agreement in Chicheŵa. *Language*, 63(4):741–782.

[Butt et al., 1999] Butt, M., Niño, M., and Segond, F. (1999). *A Grammar Writer's Cookbook*. CSLI Publications, Stanford, CA.

[Dalrymple, 2001] Dalrymple, M. (2001). *Lexical Functional Grammar*, volume 34 of *Syntax and Semantics*. Academic Press, New York.

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