XLFG Documentation

version 9.8.3 – February 2020

Lionel Clément

Contents

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 9.8.3 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, in the lexicon or in the 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, without any explanation.

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 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*.

The XLFG's front-end is a software, design for Natural Language Processing applications as Natural Language Understanding. For those interested in the back-end part, please contact the author Lionel Clément by email.

This present documentation concerns the XLFG's front-end, designed to experiment LFG grammar online for both education and research.

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

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 and Safari.

2 The web interface

The web interface is very intuitive, just go with the flow! The "How to" list may also helps you to learn about the software features.

2.1 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.

2.2 Workshops

Any recorded user can create a workshop. Workshops are designed for organize classes. A teacher or a lecturer can create a workshop, the projects related to it and assigns the rights for these projects according to the individual member whose part of the project is readable, editable or not. The goal is to share

projects or parts of them to all the membres of a workshop in order to organize a LFG lesson.

There are two ways to enrole students to a workshop: a) The lecturer invites students who have to accept the invitation to join the workshop. b) The students ask to join the workshop. The workshop's owner will receive a notification and will accept or not the application.

To find the student login name, you can use the menu `users` that is the who's who of XLFG. You will find the login names associated with the student name. However, this information is available only if the student didn't remain it secret.

2.3 Projects

A project consists in a form with a title, an optional description, an input, and output parameters which provide information for what will be the result of the parsing process.

The input consists in a set of grammatical or ungrammatical sentences, optional declarations, a grammar and a lexicon.

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

When a project is shared in 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.

Each individual copy of project is called a `version`. At any time it is possible to save the current version and to retrieve the recorded versions archived on the web.

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

- The teacher creates a workshop and invites the students to enroll. Then he produces a project including a grammar with `read-only` access and a lexicon with `read-write` access in our example.
- Each student makes a personal version of the project and may change the lexicon entries by following the teacher's guidelines without any possibility of changing anything else. They can try the parsing process, correct and save their work.
- The teacher will change to `read-only` the lexicon access to prevent any late modification by the students after a dealine. He can then quietly read and correct each student work.

The following provides a summary of access rights for each part of a shared project in a workshop. The shared data is the original project created by the teacher. The individual data are the student's versions copied from this project.

Privilege	User enrolled in the Workshop (i.e. student)
hidden	the project data is not visible, even if it is present
read-only	the project data is visible but not editable
read-write	the version data is editable

Privilege	Workshop Owner (i.e. teacher)
hidden	the project data is editable
read-only	the project data is editable
read-write	the project data and the version data are editable

Figure 1: Summary of privilege access

3 First steps with XLFG

The easiest way to get to know XLFG is to use one of the sample grammars. Choose *Simple English grammar sentence with FS* in the *Projects - English grammars* menu. A page opens with a number of pre-filled fields.

Sentences The only sentence to be parsed is *John sees a man with a telescope*, wich is ambiguous.

Declaration This part will contain some useful declarations:

The syntactic functions, the start symbol, and the morphological abbreviations.

Lexicon contains the lexical entries, written as a simple or a compound word 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.

When clicking on the *Parse* button, the output tab opens, which contains:

- A graph encoding the lattice of words sequence and all the constituents found (fig. 2).

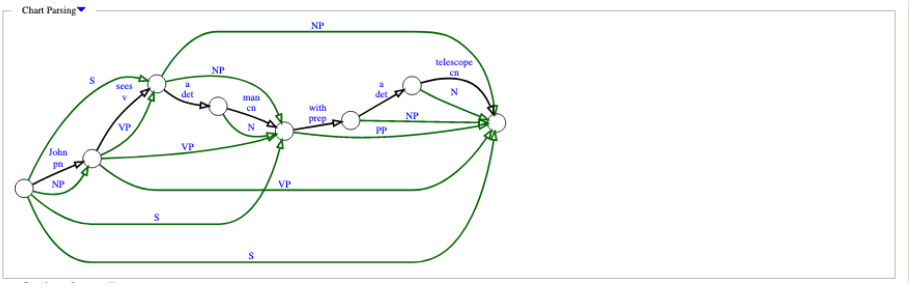


Figure 2: Lattice of words and phrases

- One of the three possible constituent-structures for the sentence (fig. 3).
- The two corresponding well-formed functional-structures (fig. 4).
- The functional equations used for this analysis (fig. 5).
- The corresponding argument-structures (fig. 6).

As the constituent-structure number can be quite large (in an exponential number given the sentence length), XLFG displays only one tree at a time, or a

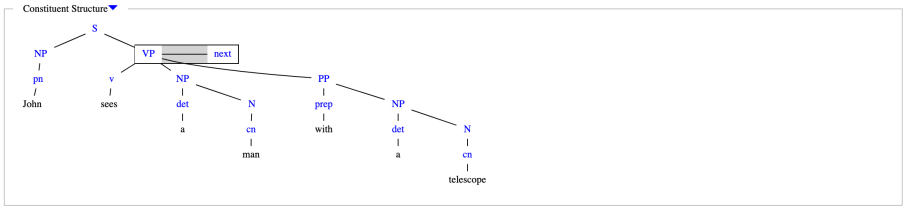


Figure 3: Constituent-Structure

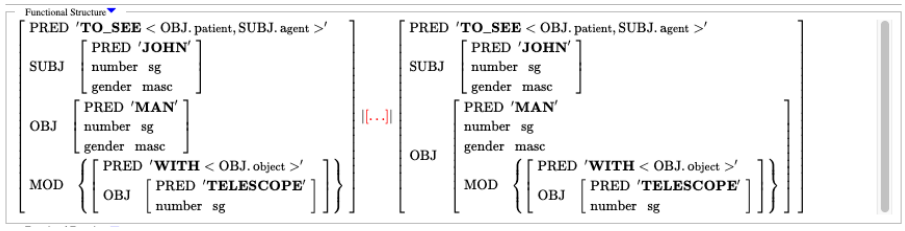


Figure 4: Functional-Structure

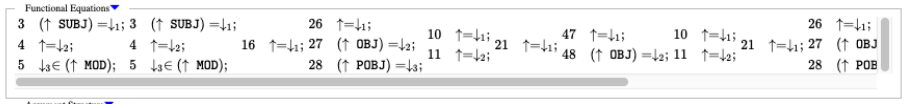


Figure 5: Functional equations

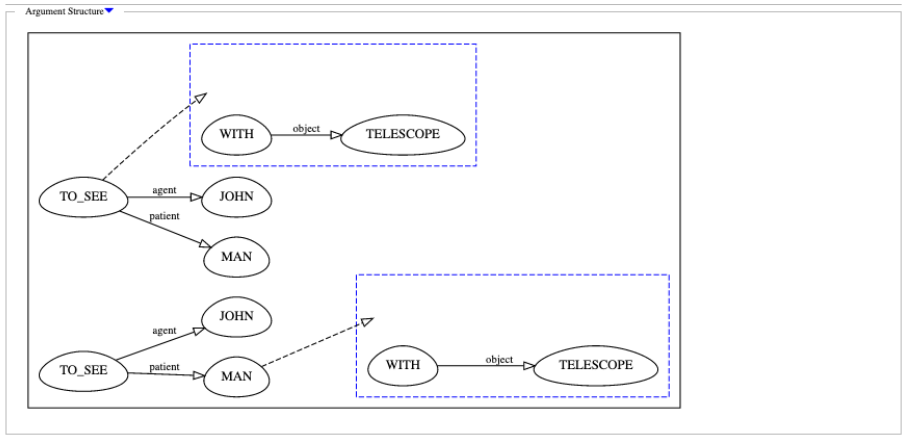


Figure 6: Argument-Structure

shared representation, namely an acyclic directed graph, or even an instantiated context-free grammar.

In our exemple, to see the other constituent-structures and their functional-

structures, click on the link labelled `next` to the `VP` node. This links bring an other possible analyse in which the phrase *with a telescope* is attached to the verb as a complement. Other click on the link `next`, and you will see the PP phrase attached to the noun.

Click on each blue links on the graphs or on the trees to see the functional-structures, functional equations and argument-structures related with it.

3.1 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*, *the men run*, **the men runs* etc. But soon these sentences will not be parsed because this first English grammar will be too poor.

3.2 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 and lexical entry to our example will allow you to parse the sentence *John sees a little man*.

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

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

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.

3.3 Error messages

If you make a mistake, you will get an error message which provides a line number and the project part where the error occurs. This will help you to go directly to the line and fix the error.

4 LFG theory and XLFG

4.1 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 modifies the clause. In both cases, the prepositional phrase (PP) *with a telescope* is the same. It is thus economical to share this part of the two analysis in the result. 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 PP modification of a noun might be written as such a nominal phrase (NP) rewriting rule:

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

In this rule, the F-structures of the noun N and the noun phrase NP 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, namely its set of equation resolution. Without such a condition, or other restrictions on constraints, LFG parsing is an undecidable problem which 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 long 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.

So, it is always possible to add constraints on embedded phrases on a rule with XLFG, but these constraints apply only locally to the rule and will never affect the embedded structure itself.

Let's take an example.

On the previous nominal phrase rule, we want to add a constraint to select which preposition is available for a noun modifier. All things being equal, this constraint will reject *the media throughout the trial* as a nominal phrase in 1a and conversely accept *the media with large audiences* in 1b.

(1) a. *The jury must not speak to the media throughout the trial.*
b. *The jury must not speak to the media with large audiences.*

For our purpose, a possible LFG rule which encodes such a constraint might look like the following

NP →	Det	N	[PP]
	↑ = ↓	↑ = ↓	↓ ∈ (↑ ADJ)
			(↓ category) $=_c$ nominal_modifier

In this rule, the scope of the constraint `(↓ category)` $=_c$ `nominal_modifier` must be limited to the `PP` embedded in the nominal phrase, but not on its other utterances, i.e. on the clause modifier. But a careful reader remembers that in XLFG, the both `PP` are only one. So, in XLFG this constraint, as all other functional descriptions, only occurs in embedded structures has a local effect. It can never be used on the embedded phrase description as a local property. In this case, for instance, it is not possible to retrieve the `category` constraint on the `PP` rule, but the `PP` may have such a `category` that is coherent with this constraint or not.

4.2 Writing XLFG rules

This is the XLFG version of the preceding rule:

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

In this declaration, the values ↑, ↓1 and ↓2, which correspond respectively to the F-structure of the `NP`, the determiner `Det` and the noun `N`, 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`.

Finally the declaration `(↓3 category) == nominal_modifier` add a constraint on local copy of the embedded `N`.

In our example, the value of the F-structure $\uparrow$ is the unification of $\downarrow 1$ and $\downarrow 2$ to which the set-valued feature `ADJ` is added.

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 and temporary recording of local F-structures.

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. The above explanation regarding the F-structure resolution algorithm is not necessary to use XLFG and describe LFG grammar within.

Formal definition of = symbol in XLFG

Given a shared forest where the node N_i dominates N_j, an equality declaration $x = y$ or $y = x$ 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.

5 XLFG F-structures

A functional structure is a set of attribute–value pairs noted $[attr_1 : val_1, attr_2 : val_2 \ldots attr_k : val_k]$. Excepted for `PRED`, `LEXEME` and `SUBCAT` which are special features that we will see later, the features are not typed, that is, their possible values are not induced by their name.

The possible values for a feature may be

- A `PRED`, `LEXEME` or `SUBCAT` (we will see later the details)
- An atom, which is an identifier without any space or special character. The maximal number of atoms per grammar is 255.
- A disjunction of atoms separated by `"|"` character,
- An F-structure,
- A disjunction of F-structures separated by `"|"` character,
- A set of features-structures marked with `"{...}"` characters,
- A litteral, which is an identifier without any space or special character marked with simple quots `'...'`. An unlimited number of literals can be used, but they can't be combined like atoms.

The following are some exemples of possible F-structures. These exemples are just written to illustrate the F-structures content, not the syntactic assumptions used.

- A F-structure that may be the result for the parsing of *John sees a man with glasses.*

```
[PRED: 'TO_SEE<SUBJ.agent, OBJ.patient>',
 SUBJ: [PRED:'JOHN',
        number: sg],
 OBJ: [PRED:'MAN',
       number: sg,
       def: false,
       MOD: { [PRED:'WITH<OBJ.object>',
               OBJ: [PRED:'GLASSES',
                     number: pl,
                     def: false ]}]]
```

- The possible lexical entry for the verbal form *snore*

[PRED: TO_SNORE<SUBJ>',
TENSE: present,
MOOD: indicative,
SUBJ: [number: sg, person: 1|2] | [number: pl]]]

- The possible lexical entry for the verbal form *kick* in the phraseological unit *to kick the bucket*

[PRED: 'TO_DIE<SUBJ.Agent>OBJ',
TENSE: present,
MOOD: indicative,
SUBJ: [number: sg, person: 1|2] | [number: pl],
OBJ: [LEXEME:'BUCKET',
number: sg,
def: true,
modified: false]]

- The possible lexical entry for the verbal form *look* in the verbal phrase *to look for*

[PRED: 'TO_LOOK_FOR<SUBJ.Agent, OBJ.Theme>',
TENSE: present,
MOOD: indicative,
SUBJ: [number: sg, person: 1|2] | [number: pl],
OBJ:[prep_form:'for']]

5.1 A F-structure as a set of F-structures

([Kaplan and Maxwell, 1988]) proposed an unstructured sets of F-structures used as the functional representation of coordinate constructions.

The rule used to construct such a set is the following:

$$\begin{array}{lllll} V \rightarrow & V & \text{conj} & V \\ & \downarrow \in \uparrow & & \downarrow \in \uparrow \end{array}$$

They also indicated the paradox between a functional representation of the conjunction and such a set construction. If we add $\uparrow = \downarrow$ to the previous rule, the conjunction will be distributed to the embedded elements and the F-structure as a whole will be inconsistent. The solution adopted by ([Kaplan and Maxwell, 1988])

was to do not represent a PRED for coordinations, but a semantic projection without PRED.

In ([Clément, 2019]), we defend the idea of reclaiming `PRED` as the predicate of coordination structures and we discarded the possibility of incoherent result with unlike conjunctions.

So the following rule is available in XLFG:

V → V conj V {
↓1 ∈ ↑;
↓2 = ↑;
↓3 ∈ ↑; }

In XLFG, the formal definition of F-structure has been modified. It is no longer only a set of features, it can also contain a set of F-structures.

The following shows an extract of such a F-structure which contains a set of F-structures for the analysis of *Lucy can finish her work and John its shopping*:

```
[PRED: 'AND',
 { [PRED: 'CAN_FINISH<SUBJ.agent, OBJ.patient>',
   SUBJ: [PRED: 'LUCY', ... ],
   OBJ: [PRED: 'WORK', ... ],
   ... ],

   [PRED: 'CAN_FINISH<SUBJ.agent, OBJ.patient>',
   SUBJ: [PRED: 'JOHN', ... ],
   OBJ: [PRED: 'SHOPPING', ... ],
   ... ]}]
```

5.2 Shared F-structures

As well as constituent-structures, the F-structures which result of a parsing are in an exponential number. In order to construct a polynomial algorithm upon the length of the sentence with LFG theory, XLFG represents a compact version of F-structures.

We therefore need a new definition of a F-structure:

- A set of features, that are the shared features;
- A set of F-Structures written with {} characters, that is the coordinated F-structures (also shared);
- A set of F-Structures written without {} delimiters and separated with the "`|`" character. This set defines the distributive part of the F-structure. Since each F-structure may contains such a set of F-structure, an exponential number of F-structures may be described in this way.

The figure 7 is the shared F-structure that result of the parsing of *John sees a man with a telescope*, where the subject is shared between several analysis:

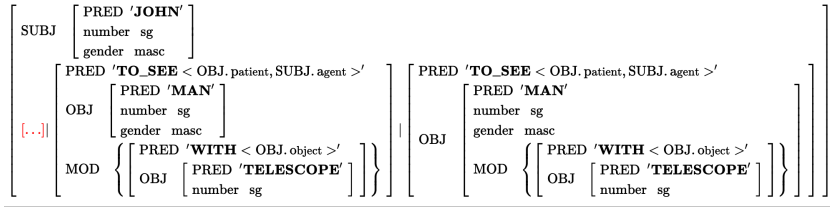


Figure 7: The shared F-structure for *John sees a man with a telescope*

5.3 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.

(2) 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.

(3) 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.

(4) 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 *to 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.

(5) 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.

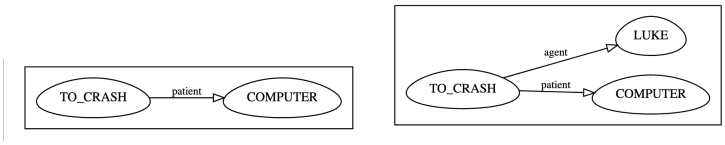


Figure 8: The computer crashed – Luke crashed the computer

5.4 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:

(6) 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 simplified F-structure of 6a is:

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

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

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

By contrast, the F-structure of 6b 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 co-indexed semantically is a separate issue that we do not model here).

Now let us consider what the F-structure of the agrammatical sentence 6c. 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 lexeme is the combination of the two lexemes.

PRED: ' lexeme < . . . > . . . '
LEXEME: ' prefix - '
LEXEME: ' - 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*.

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 first tentative result for the lexical entries are as follow (the XLFG syntax of this tentative is not correct, a `PRED` feature doesn't support prefix or suffix lexemes, see later the `LEXEME` feature).

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 two PRED attributes in XLFG:

	Prefix	Suffix	Lexeme
Prefix	None	None	Lexeme
Suffix	None	None	Lexeme
Lexeme	Lexeme	Lexeme	$\perp$

As one can see, it is always possible to unify more than two PRED, thanks to the idempotence property of unification of a lexeme with a prefix or suffix lexeme. This possibility may happily be used for serial verb analysis, taking care of operation's order.

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 ([Clément and Diao, 2016]).

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 particule *up*. Moreover, the subcategorization of *to give up* doesn't result on the combination between the lexical entries of the verb *to give* and the adverb *up*. So the previous combination between `to give` and `lecture` which resulted in the `PRED`

'GIVE_LECTURE<SUBJ, [onOBL]>' is not an universal rule which may be systematically applied.

(7) 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 validate this new lexical entry from such a combination, XLFG uses 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]>',
        mode: indicative,
        tense: present];
up VERB_PART [LEXEME:' - _UP',
                    part_form: 'up'];
```

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

- the lexeme is the combination between the two lexemes `TO_GIVE` and `-_UP`, that is "`TO_GIVE_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] >',
 mode: indicative,
 tense: present,
 part_form: 'up'];
```

In XLFG, each complex lexical entry is explicitly described with a `#` that adds the sub-categorization, and some specific features for the complex lexical entry.

This explicit lexical entries are also useful to detect the correct lexical combinations and reject some unspeakable constructions between verbs and particules.

5.5 The feature `LEXEME`

By contrast, idiomatic syntactic constructions are not the combination of several lexemes, but a specific lexical entry for an head which constraints its complements.

(8) a. *Mary kicks the bucket.*
b. *Mary kicks the buckets.*
c. *Mary kicks a bucket.*
d. *Mary kicks a yellow bucket.*
e. *Mary kicks a bucket of water.*
f. *Mary kicks the habit of smoking.*
g. *How to naturally kick the smoking habit?*

In 11a, the semantic value of the phraseme *to kick the bucket* is *to die*, but the multi-word expression is very constrained: the complement *bucket* must be singular, not modified, without any complement and definite, as one can see in 8b – 8e

One possible lexical entry for the form *to kick the bucket* witch contains the lexeme `bucket` and its constraints is the following:

```
kicks [PRED: 'TO_DIE < SUBJ> OBJ',
       mode: indicative,
       tense: present,
       OBJ: [LEXEME: 'BUCKET',
             number: sg,
             def: true,
             mod: false];
```

In 8f, , the phraseme *to kick the habit of sth* is constrained with the lexeme `habit` and other constraints, but, furthermore, its sub-categorization is not a combination between the sub-categorizations of its parts.

A possible lexical entry for the form *kicks* is the following:

```
kicks v [PRED:'TO_KICK_THE_HABIT<SUBJ.AGENT,
                                 [OFVing.THEMA]>OBJ',
        OBJ:[LEXEME:'HABIT', number:sg,
             mod:false, def: true]]
        {
        (↑ OFVing SUBJ) = (↑ SUBJ);
        }
```

An F-structure that contains a `LEXEME` feature is not completed, which means that such a F-structure must be unified with another F-structure to build a `PRED` feature in place of it. Considering this constraint, the lexical entry *kicks* for the word phrase *to kick to habit of doing sth* is not possible without the form *habit* as an object.

5.6 The feature `SUBCAT`

If they want to add a lexical entry without any lexeme but with a constraint on sub-categorization. It can be added a `SUBCAT` feature with only this property. In this case the feature `SUBCAT` can unified with any `PRED` without a combination of lexemes.

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

	PRED:X<Y>Z	LEXEME:X	LEXEME:X-	LEXEME:-X	SUBCAT:<Y>Z
PRED:A<B>C	⊥	⊥	PRED:XA<B>C	PRED:AX<B>C	PRED:A<B U Y>C U Z
LEXEME: A	⊥	⊥	LEXEME:XA	LEXEME:AX	PRED:A<Y>Z
LEXEME: A-	PRED:AX<Y>Z	LEXEME:AX	⊥	⊥	⊥
LEXEME: -A	PRED:XA<Y>Z	LEXEME:XA	⊥	⊥	⊥
SUBCAT: <B>C	PRED:X<B U Y>C U Z	PRED:X<B>C	⊥	⊥	SUBCAT:<B U Y>C U Z

In this table, `X U A` refers to the unification between lexemes that is possible (if one of them is a prefix or a suffix) or not. ⊥ (bottom symbol) refers to an unification failure.

6 Functional descriptions

The Functional description is a crucial part of the LFG theory. It defines the Φ projection allowing one to construct a F-structure from a C-structure. They also provide explicit constraints on the resulting F-structures.

Syntactic relations, local and nonlocal agreement, subject control, sub-categorization, and many other syntactic phenomena can be modeled using functional descriptions. The following examples are only illustrative, and the interested reader is directed to the LFG literature for further reading and analyses.

6.1 Functional equations

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:

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

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

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

6.2 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

```
↓i ∈ (↑ <path>)
(↓i <path>) ∈ (↑ <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);
```

$\uparrow = \downarrow 2$;
};

N $\rightarrow$ N REL
{
$\uparrow = \downarrow 1$;
$\downarrow 2 \in (\uparrow \text{ADJ})$;
};

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

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

6.3 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:

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

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.

XLFG will automatically detects such a situation and will create a link to the subject (↑ SUBJ).

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.

6.4 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 $\left[\ \text{CPLZER} \quad \text{THAT}\ \right]$, 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;
   (↓1 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.

6.5 Negative constraints

The operator $\neq$ 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> $\neq$ <constant>

This constraint fails if and only if the constants exists with values that matches.

6.6 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.

(9) 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.

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

```
to PART [] {¬(↑ TENSE);};
```

6.7 Conditionals

Conditionals on C-structure

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 (¬$i) <statement>
if (¬$i) <statement> else <statement>
```

where $i is true if and only if the i^{th} term of the rule is present.

Conditionals on F-structure

If a constraining equation, a negative constraint or an existential constraint is the condition to evaluate some other constraints or functional descriptions, the operator if is also available to do so:

```
if (<expression>) <statement>
if (<expression>) <statement> else <statement>
```

For exemple, in French, the gender of coordinated nouns depends on the gender of its parts: to be marked as feminine, all the parts have to be in feminine gender. While the person is marked as 1st (resp. 2nd) when just one part is at the 1st (resp. 2nd) person.

(11) a. *Jean et moi sommes partis*
Jean (masc) and me (fem/masc) went (1st - plural - masc)

b. *Jeanne et moi sommes parties/partis*
Jean (fem) and me (fem/masc) went (1st - plural - fem/masc)

c. *Jeanne et toi êtes parties/partis*
Jean (fem) and me (fem/masc) went (2nd - plural - fem/masc)

d. *Jeanne et Michel sont partis*
Jeanne (fem) and Michel (masc) went (3rd - plural - masc)

e. *Jeanne et Michèle sont parties*
Jeanne (fem) and Michèle (fem) went (3rd - plural - fem)

Following these guidelines, the XLFG rule for `NP` coordination in French is as simple as this:

```
NP → NP coo NP {
        (↑ number) = pl;

        if (((↓1 person) == 1) ∨ ((↓3 person) == 1))
          (↑ person) = 1;
        else if (((↓1 person) == 2) ∨ ((↓3 person) == 2))
          (↑ person) = 2;
        else
          (↑ person) = 3;

        if (((↓1 gender) == fem) ∧ ((↓3 gender) == fem))
          (↑ gender) = fem;
        else
          (↑ gender) = masc;

        ↓1 ∈ ↑;
        ↑ = ↓2;
        ↓3 ∈ ↑;
}
```

6.8 Selection

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

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

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

The equivalent XLFG notation for term selection, is written by using the "|" symbol.

```
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>
   ...
}
```

6.9 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.

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

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;
}
```

6.10 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)`, `A*`, `A+`, corresponding respectively to concatenation, disjunction, iterative closure, and iterative closure with at least one element.

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

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

(↑ (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 the structures y, w.

6.11 The cut operator "!"

Without any pragmatic context, some sentences are ambiguous between a phrasal expression and a litteral expression.

(12) a. *The boat began to sway,* ***broke the ice*** *which was surrounding it, and began to move forward slowly.*

b. *This simple gesture* ***broke the ice****.*

As you have probably noticed by now, XLFG was designed to preserve all the ambiguities between each level of the analysis. It keeps shared structures during the process as a whole.

The meaning of an ambiguous sentence like 12a, 12b is based on a pragmatic analysis, that is not involved in the XLFG syntactic process.

In fact, in most cases it's unlikely that they will be. The phrase words and the compounds are almost always the right lexical entry, and you can drop the rest that it is statistically less probable.

There are two ways in XLFG to simulate such a probability. The first one is to add a new feature with a weight level on each word. This rank may be given by a statistical study on texts and retrieved at the end of the parsing to rank the output results. The second one, is to select only the phrase word and never select the others. An XLFG operator "!" is provided for that purpose: in a lexical rule or in a syntactic rule, one may add this equation to present the other analysis:

↑ = !;

A possible XLFG lexical entry for the form *broke* which allows us to have a correct analysis of the sentences 12a, 12b is the following:

```
broke [PRED: 'TO_BRAKE_THE_ICE<SUBJ.Agent>OBJ',
       TENSE: past,
       MOOD: indicative,
       OBJ: [LEXEME:'ICE',
             number: sg,
             def: true,
             modified: false]]{
       ↑ = !;
}
|[PRED: 'TO_BRAKE<SUBJ.Agent, OBJ.Patient>',
  TENSE: past,
  MOOD: indicative]
```

In 12a, all the constraints of the first entry are applied, and the "!" operator reject the second entry.

In 12b, the noun *ice* is modified with the relative *which was surrounding it* that is incompatible with the feature `modified:false`. The only entry which is compatible is the second one.

7 Constraints on F-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.

7.1 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.*

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

7.2 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.

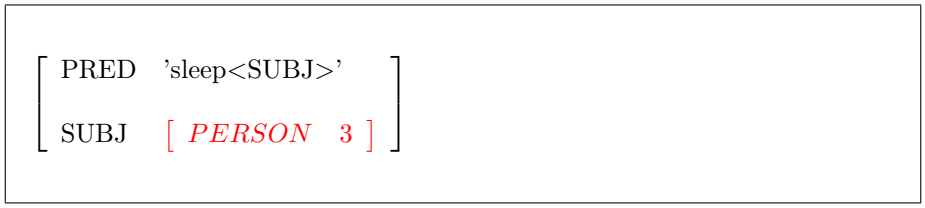
Here is an example of an incomplete F-structure.
**Mary precedes.*

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

7.3 Extended Coherence

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

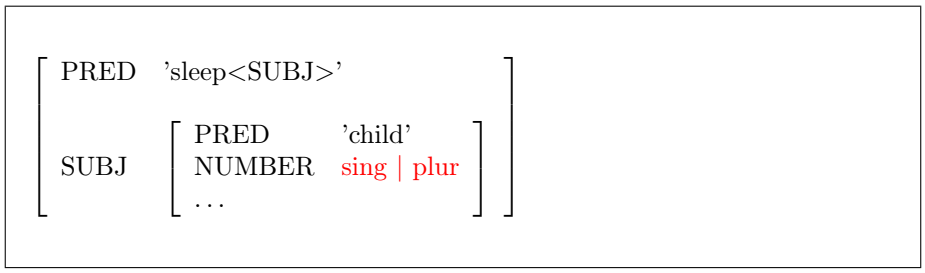
Here is an example of an *extended* incoherent F-structure:
**sleeps well.*



7.4 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.*



8 Technical XLFG specifications

An XLFG grammar consists in 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.

8.1 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.

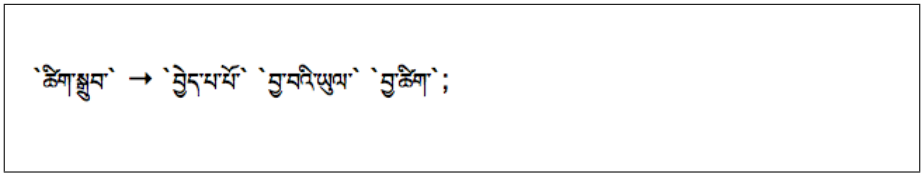
- The latin characters are a ... z, A ... Z, à, á, â, ã, ä, å, æ, ç, è, é, ê, ë, ì, í, î, ï, ð, ñ, ò, ó, ô, õ, ö, ø, ù, ú, û, ü, ý, ÿ, θ, À, Á, Â, Ã, Ä, Å, Æ, Ç, È, É, Ë, Ì, Í, Î, Ï, Ð, Ñ, Ò, Ó, Ô, Õ, Ö, Ø, Ù, Ú, Û, Ü, Ý, Ÿ, Þ, ß
- The keywords are the following:

- `start_symbol` or `start symbol`
- `syntactic_functions` or `syntactic functions`
- `PRED`
- `LEXEME`
- `SUBCAT`
- `switch`
- `case`
- `if`
- `else`
- `with`
- `in`
- `lex`

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:



The maximal number of different identifiers in the same project is limited to 255, witch is suffisant for building large coverage XLFG grammars for many languages.

8.2 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:

$VP \rightarrow V\ PP^*$

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 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 keyword `start_symbol` in the declaration section of the project.

```
start symbol: NP;
```

8.3 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]]
```

8.3.1 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 255.

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

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

The number of literals is not limited.

8.3.2 Embedded feature-structures

Feature values for grammatical functions are embedded feature-structures

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

8.3.3 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.

8.4 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]
];
```

9 XLFG lexicon

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@u-bordeaux.fr`, the second matches an url as `https://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 $\downarrow i$.

9.1 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_'];
```

9.2 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];
```

9.3 The lex function

In [Clément, 2019], we added a new operator in XLFG witch does'n exist in LFG theory: the *lexical capture*. The lexical path ($\downarrow$1 **lex**) allows us to re-

trieve all lexical entries based on the local lexeme (i.e. the PRED feature) of ↓1. These lexical entries are not only the same word, but also different forms according to its morphology like is the sentence 13a. In more difficult cases, it can be a combination between several lexemes (light verbs, phrasemes, serial verbs, etc). This explains for exemple the zeugma constructions that we have formalized with XLFG. See this example where the light verb *to execute (a law)* is coordinated with the plain verb *to execute sb* in the second example (13b): Furthermore, there is no reason to think that the lexical entry retrieved must have the same sub-categorisation as the original one.

(13) a. *Lucy has to finish high school and her brothers their Master Degrees.*
b. *You are free to execute your laws, and your citizens, as you see fit* (Star Trek: *The Next Generation*)

In order to add such new lexical entries with a given lexeme and a new sub-categorization, one has to add the following line in the lexicon wich will contains at least the sub-categorization:

#<lexeme> <features> [<local functional descriptions>];

In 13a, the sequence *her brother its Master Degree* isn't a sentence according to the fact that it doesn't contain any verb. In fact, the verb is in the first part of the coordination.

But we see that a) The verb form must be *have to*, not *has to*, according to the morphological agreement with a plural noun. b) the verb is composed with the auxiliary *have to finish*, it is not the simple form *finish*.

If in our grammar, the local predicate of the sentence *Lucy has to finish high school* is

PRED: 'HAVE_TO_FINISH<SUBJECT.agent, OBJECT.patient>'

We then add the following lexical entry in the lexicon:

#HAVE_TO_FINISH [SUBCAT:'<SUBJECT.agent, OBJECT.patient>'

The F-structure resulting to the parsing is the following:

```
[PRED: 'AND',
{
   [PRED:'HAVE_TO_FINISH<SUBJECT.agent, OBJECT.patient>',
    SUBJECT:[PRED:'LUCY', ... ],
    OBJECT:[PRED:'HIGH_SCHOOL', ...]
   ],
   [PRED:HAVE_TO_FINISH<SUBJECT.agent, OBJECT.patient>',
    SUBJECT:[PRED:'BROTHER', ...],
    OBJECT:[PRED:'MASTER DEGREE', ...]
   ] }
```

In the second example, the lexeme is not the same; the first part of the coordination, it is a light verb construction *execute a law*, when it is the plain verb *to execute* in the second part.

The lexical entries are the following:

```
executeVERB[PRED: 'TO_EXECUTE<SUBJECT.Agent, OBJECT.Patient>']
       | VERB[LEXEME: 'EXECUTE_-'];
laws NOUN[LEXEME: 'LAW'];
# TO_EXECUTE_LAW [SUBCAT: '<SUBJECT.Agent , OBJECT.Theme>'];
```

The F-structure resulting to the parsing is the following:

```
[PRED: 'AND',
{
   [PRED:'EXECUTE_LAW<SUBJECT.Agent, OBJECT.Theme>',
    SUBJECT:[1][PRED:'YOU', ... ],
    OBJECT:[PRED:'LAW', ...]
   ],
   [PRED:EXECUTE<SUBJECT.Agent, OBJECT.Patient>',
    SUBJECT:[1],
    OBJECT:[PRED:'CITIZEN', ...]
   ] }
```

The thematic relation of *to execute a law* is not exactly the same as *to execute sb.* Indeed, a law is not a patient, but a theme, because it does not change its state when it is executed. A person who is executed unfortunately does.

10 ANNEXE 1 – XLFG Language

10.1 The statements (<stm>)

The Functional equations

↑ = ↓i
↓i = ↑
(↓i <path>) = <atom>
(↓i <path>) = <literal>
(↓j <path>) = ↓i
(↓j <path>) = (↓i <path>)
(↓i <path>) = <features>
↓i = <features>
(↑ <path>) = <atom>
(↑ <path>) = <literal>
(↑ <path> **LEXEME**) = <literal>
(↑ <path> **PRED**) = <pred>
(↑ <path> **SUBCAT**) = <subcat>
$id = ↓i
(↑ <path>) = ↓i
↑ = (↓i <path>)
$id = (↓i <path>)
(↑ <path>) = (↓i <path>)
(↓i <path>) = (↑ <path>)
↑ = <features>
↑ = !
$id = <features>
(↑ <path>) = <features>
$id = (↑ <path>)
$id = ($id <path>)
(↑ <path>) = (↑ <path>)

The in-set descriptions

↓i ∈ ↑
↓i ∈ $id
↓i ∈ (↑ <path>)
↓i ∈ (↓ <path>)
(↓i <path>) ∈ ↑

(↓i <path>) ∈ $id
(↓i <path>) ∈ (↑ <path>)
(↓i <path>) ∈ (↓ <path>)
<features> ∈ ↑
<features> ∈ $id
<features> ∈ (↑|↓ <path>)

The conditional descriptions depending on C-structures

if ($i) <stm>
if ($i) <stm> **else** <stm>
if (¬$i) <stm>
if (¬$i) <stm> **else** <stm>

The constraints (<expression>)

<path> ≠ <atom>
<path> ≠ <literal>
(↑ <path> **LEXEME**) ≠ <literal>
<path> ≠ <path>
<path> == <atom>
<path> == <literal>
(↑ <path> **LEXEME**) == <lexeme>
<path> == <path>
<path>
(↑ <path> **LEXEME**)
¬<path>
¬(↑ <path> **LEXEME**)

The conditional descriptions depending on F-structures

if (<boolean_expression>) <stm>
if (<boolean_expression>) <stm> **else** <stm>

<boolean_expression> ::=
(<expression>)
<boolean_expression> ∧ <boolean_expression>
<boolean_expression> ∨ <boolean_expression>
<boolean_expression> ⇒ <boolean_expression>
<boolean_expression> ⇔ <boolean_expression>

The case descriptions

switch ($i) {<list_case>}

<list_case> ::= **case** <id>:<stm> <list_case>
|**case** <id>:<stm>

The long-distance dependencies

with $id **in** (↑ <regexp>) <stm>

10.2 Features description (<features>)

[. . .]
[]
(**lex** ↑)
(**lex** ↓i)
(**lex** (↑ <path>))
(**lex** (↓i <path>))

11 ANNEXE 2 – The full XLFG grammar

```
axiom: script_lines

script_lines: script_line
            | script_lines script_line

script_line: _grammar_ "≪" rules "≫"
           | _lexicon_ "≪" dictionary "≫"
           | _declarations_ "≪" declarations "≫"
           | _entries_ entries ";"

declarations: declarations declaration
            | /* empty */

declaration: "@" IDENTIFIER ":" list_feature ";"
           | "start_symbol" ":" term_id ";"
           | "start" "symbol" ":" term_id ";"
           | "syntactic_functions" ":" defFunctions ";"
           | "syntactic" "functions" ":" defFunctions ";"
           | IDENTIFIER ":" type ";" (not documented)
           | error ";"

type: "[" "]"
    | "{" "}"
    | atom

defFunctions: defFunctions "," IDENTIFIER
            | IDENTIFIER
            | /* empty */

rules: rule rules
     | rule

rule: term_id "→" terms_vector statements_or_semi

rule: term_id "→" statements_or_semi

terms_vector: terms_vector terms
            | terms

terms: terms_disj
     | "[" terms_disj "]"
```

```
terms_disj: terms_disj "|" term_id
          | term_id

term_id: IDENTIFIER

dictionary: lexicon_line dictionary
          | lexicon_line

lexicon_line: IDENTIFIER entries ";"
            | STRING entries ";"
            | "#" IDENTIFIER features_opt ";"
            | error ;

entries: entry "|" entries
       | entry

entry: "(" entry ")"
     | entry "&" arg
     | arg

arg: IDENTIFIER features_opt

features_opt: features
            | /* empty */

features: "[" list_feature "]" statements_opt

features: "[" "]" statements_opt

list_feature: list_feature "," feature
            | feature
            | list_feature "," list_features_pipe_plus
            | list_features_pipe_plus
            | list_feature "," "{" list_features_comma "}"
            | "{" list_features_comma "}"
            | list_feature "," "@" IDENTIFIER
            | "@" IDENTIFIER

feature: "PRED" ":" "'" IDENTIFIER subcat "'"
       | "PRED" ":" "'" IDENTIFIER "-" subcat "'"
       | "PRED" ":" "'" "-" IDENTIFIER subcat "'"
       | "LEXEME" ":" "'" "-" IDENTIFIER "'"
       | "LEXEME" ":" "'" IDENTIFIER "-" "'"
```

```
        | "LEXEME" ":" "'" IDENTIFIER "'"
        | "SUBCAT" ":" "'" subcat "'"
        | atomIdentif ":" features
        | atomIdentif ":" list_features_pipe_plus
        | atomIdentif ":" atom
        | atomIdentif ":" "'" IDENTIFIER "'"
        | atomIdentif ":" "{" list_features_comma "}"

list_features_comma: list_features_comma , features
                   | features

list_features_pipe_plus: list_features_pipe "|" features

list_features_pipe: list_features_pipe "|" features
                  | features

subcat: "<" functions_opt ">" functions_without_actors_opt
      | /* empty */

functions_opt: functions
             | /* empty */

functions: functions "," function
         | function

functions_without_actors_opt: functions_without_actors
                            | /* empty */

functions_without_actors: functions_without_actors,
                          function_without_actors
                          | function_without_actors

function: function_without_actors
        | atomIdentif "." atomUniqIdentif
        | "[" atomIdentif "." atomUniqIdentif "]"

function_without_actors: atomIdentif
                       | "[" atomIdentif "]"

atom: atomUniq
    | atomUniq "|" atom

atomIdentif: atomUniqIdentif
           | atomUniqIdentif "|" atomIdentif
```

```
atomUniq: atomUniqSign
        | atomUniqIdentif

atomUniqIdentif: IDENTIFIER

atomUniqSign: INTEGER
            | "+"
            | "-"

statements_base: "{" list_statement "}"
               | "{" "}"

statements_opt: statements_base
              | /* empty */

statements_or_semi: statements_base
                  | ";"

list_statement: statement
              | list_statement statement

statement: expression ";"
         | "{" list_statement "}"
         | "{" "}"
         | "↑" "=" down_stm ";"
         | down_stm "=" "↑" ";"
         | down_path_stm "=" atom_stm ";"
         | down_path_stm "=" literal_stm ";"
         | down_path_stm "=" down_stm ";"
         | down_path_stm "=" down_path_stm ";"
         | down_path_stm "=" features_stm ";"
         | down_stm "=" features_stm ";"
         | up_path_stm "=" atom_stm ";"
         | up_path_stm "=" literal_stm ";"
         | up_path_lexeme_stm "=" lexeme_stm ";"
         | up_path_pred_stm "=" pred_stm ";"
         | up_path_subcat_stm "=" subcat_stm ";"
         | $ IDENTIFIER "=" down_stm ";"
         | up_path_stm "=" down_stm ";"
         | "↑" "=" down_path_stm ";"
         | $ IDENTIFIER "=" down_path_stm ";"
         | up_path_stm "=" down_path_stm ";"
         | down_path_stm "=" up_path_stm ";"
```

```
            | "↑" "=" features_stm ";"
            | "↑" "=" "!" ";"
            | "$" IDENTIFIER "=" features_stm ";"
            | up_path_stm "=" features_stm ";"
            | "$" IDENTIFIER "=" up_path_stm ";"
            | up_path_stm "=" up_path_stm ";"
            | down_stm "∈" "↑" ";"
            | down_stm "∈" "$" IDENTIFIER ";"
            | down_stm "∈" path_stm ";"
            | down_path_stm "∈" "↑" ";"
            | down_path_stm "∈" $ IDENTIFIER ";"
            | down_path_stm "∈" path_stm ";"
            | features_stm "∈" "↑" ";"
            | features_stm "∈" $ IDENTIFIER ";"
            | features_stm "∈" path_stm ";"
            | "if" ( dollar_stm ) statement
            | "if" ( dollar_stm ) statement "else" statement
            | "if" ( ¬dollar_stm ) statement
            | "if" ( ¬dollar_stm ) statement "else" statement
            | "switch" ( dollar_stm ) "{" list_case_statement "}"
            | "with" var_stm in "(" "↑" regExpE ")" statement
            | "if" "(" boolean_expression ")" statement
            | "if" "(" boolean_expression ")" statement "else" statement

expression: path_stm "≠" atom_stm
            | path_stm "≠" literal_stm
            | up_path_lexeme_stm "≠" lexeme_stm
            | path_stm "≠" path_stm
            | path_stm "==" atom_stm
            | path_stm "==" literal_stm
            | up_path_lexeme_stm "==" lexeme_stm
            | path_stm "==" path_stm
            | path_stm
            | up_path_lexeme_stm ";"
            | "¬" path_stm
            | "¬" up_path_lexeme_stm

boolean_expression: "(" boolean_expression ")"
                  | boolean_expression "∧" boolean_expression
                  | boolean_expression "∨" boolean_expression
                  | boolean_expression "⇒" boolean_expression
                  | boolean_expression "⇔" boolean_expression
                  | expression
```

```
list_case_statement: case_statement list_case_statement
                   | case_statement

case_statement: "case" IDENTIFIER ":" statement

atom_stm: atom

pred_stm: path_pred_stm
        | "'" IDENTIFIER subcat '
        | "'" IDENTIFIER "-" subcat '
        | "'" "-" IDENTIFIER subcat '

lexeme_stm: "'" IDENTIFIER "'"
          | "'" IDENTIFIER "-" "'"
          | "'" "-" IDENTIFIER "'"

literal_stm: "'" IDENTIFIER "'"

subcat_stm: "'" subcat "'"

features_stm: features
            | "(" lex up_stm ")"
            | "(" lex down_stm ")"
            | "(" lex up_path_stm ")"
            | "(" lex down_path_stm ")"

path_stm: up_path_stm
        | down_path_stm

up_path_stm: "(" up_stm apply_path_cdr_stm ")"

down_path_pred_stm: "(" down_stm apply_path_cdr_pred_stm ")"

up_path_pred_stm: "(" up_stm apply_path_cdr_pred_stm ")"

path_pred_stm: up_path_pred_stm
             | down_path_pred_stm

up_path_lexeme_stm: "(" up_stm apply_path_cdr_lexeme_stm ")"

up_path_subcat_stm: "(" up_stm apply_path_cdr_subcat_stm ")"

down_path_stm: "(" down_stm apply_path_cdr_stm ")"
```

var_stm: "$" **IDENTIFIER**

up_stm: "↑"
| "$" **IDENTIFIER**

down_stm: "↓"
| "↓" **INTEGER**
| "#" **INTEGER** (not documented)

dollar_stm: "$" INTEGER

apply_path_cdr_stm: apply_step_path_stm last_step_path_stm
| last_step_path_stm

apply_path_cdr_pred_stm: apply_step_path_stm **PRED**
| **PRED**

apply_path_cdr_lexeme_stm: apply_step_path_stm **LEXEME**
| **LEXEME**

apply_path_cdr_subcat_stm: apply_step_path_stm **SUBCAT**
| **SUBCAT**

apply_step_path_stm: apply_step_path_stm step_path_stm
| step_path_stm

step_path_stm: atomIdentif
| down_path_stm

last_step_path_stm: atomIdentif
| down_path_stm
| step_path_stm "-" step_path_stm

regExpE: regExpE regExpT
| regExpT

regExpT: regExpT "|" regExpF
| regExpF

regExpF: "(" regExpE")"
| regExpF "*"
| regExpF "+"
| regExpF "?"
| atomUniqIdentif

12 ANNEXE 3 – Some technical aspects of the XLFG software

Copyright

Actual version: 9.8.3

Major 9: The first version was created in 1996 in Lisp language, the second one in Pascal language, the third in C language. It was exponential parsers with `TCL/TK`, then `GTK` interface that I developed when I was a Master student. The 9th version is a version developed in C++ that began in 2012, with full shared structures, and an algorithm which is polynomial in the size of the parsed text (with some restrictions). Among other things, it can deal with a compacted realistic lexicon.

Minor 8: The previous minor-versions of XLFG language are no longer compatible with it. The current documentation belongs to this minor version.

Micro 3: Some bugs was fixed.

12.1 Back-end software

Some algorithms adopted for this software

- Regular expressions → deterministic automaton to split the input text in simple words and compounds as a finite-state machine (FSM).
- LALR parsing to read the input parameters, that are the XLFG grammar, the local XLFG lexicon, and the XLFG description (using the `lex`/`yacc` compiler-compiler).
- Building a FSM with the big lexicon → deterministic automaton → compacted data (in delay-time). This part of the software is not available in the front-end.
- Adapted and optimized Earley algorithm to build a shared forest from the input FSM.
- Resolution of operational semantics in order to interpret the functional descriptions on each node of the shared forest.

- Solving functional descriptions with unification. Adapted and optimized unification algorithm: low-level bitset operations and tabulation of shared results.
- Construction of FSM with regular expressions for treatment of long distance dependencies.

Complexity of the algorithms

- Input text in size $n \rightarrow$ FSM. $O(n)$
- XLFG grammar, local XLFG lexicon, XLFG description $\rightarrow$ intern representations. $O(n)$
- Input text $\rightarrow$ shared C-structure. $O(n^3)$
- Input text $\rightarrow$ shared F-structure. $O(n^q)$, q constant (without composition of long distance dependencies that never occurs in realistic grammars).
- Shared C-structures and F-structures $\rightarrow$ list of C-structures and F-structures one by one. $O(q^n)$

C_{++} code of back-end software

- 22k lines, 41 classes in C_{++} language
- 41 `.h` files, 41 `.cc` files, 2 `lex` files, 1 `yacc` file

12.2 Front-end software

PHP and JS code

- MVC, PHP object programming
- Recording the web data with `MySQL` and `Subversion`
- 32k lines, 83 classes in PHP language
- 142 `.php` files, 34 `.js` files

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.

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

[Clément, 2019] Clément, L. (2019). Une etude de la coordination des propositions avec ellipse en français : formalisation et application avec XLFG. *Langue française*. `https://hal.archives-ouvertes.fr/hal-02374592`.

[Clément and Diao, 2016] Clément, L. and Diao, S. (2016). For a unified treatment of particle verbs (Headlex16). Joint 2016 Conference on Head-driven Phrase Structure Grammar and Lexical Functional Grammar. `https://hal.archives-ouvertes.fr/hal-02489993`.

[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.

[Kaplan and Maxwell, 1988] Kaplan, R. M. and Maxwell, J. T. (1988). Constituent coordination in Lexical-Functional Grammar. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING '88)*, pages 303–305, Budapest. Reprinted in Mary Dalrymple, Ronald M. Kaplan, John Maxwell, and Annie Zaenen, eds., *Formal Issues in Lexical-Functional Grammar*, 199–210. Stanford: CSLI Publications. 1995.