UNITEX

USER MANUAL

Université de Marne-la-Vallée

http://www-igm.univ-mlv.fr/~unitex
unitex@univ-mlv.fr

Sébastien Paumier - January 2003

English translation by the local grammar group at the
CIS, Ludwig-Maximilians-Universität, Munich - Oct 2003
(Wolfgang Flury, Franz Guenthner, Friederike Malchok, Clemens Marschner, Sebastian Nagel, Johannes Stiehler)
http://www.cis.uni-muenchen.de/

Contents

Introduction

Unitex is a collection of programs developped for the analysis of texts in natural languages by using linguistic resources and tools. These resources consist of electronic dictionaries, grammars and lexical grammar tables, initially developed for French by Maurice Gross and his students at the Laboratoire d'Automatique Documentaire et Linguistique (LADL) . Similar resources have been developed for other languages in the context of the RELEX laboratory network.

The electronic dictionaries specify the simple and compound words of a language together with their lemmas and a set of grammatical (semantic and inflectional) codes. The availability of these dictionaries is a major advantage compared to the usual utilities for pattern searching as the information they contain can be used for searching and matching, thus describing large classes of words using very simple patterns. The dictionaries are presented in the DELA formalism and were constructed by teams of linguists for several languages (French, English, Greek, Italian, Spanish, German, Thai, Korean, Polish, Norwegian, Portuguese, etc.)

The grammars deployed here are representations of linguistic phenomena on the basis of recursive transition networks (RTN), a formalism closely related to finite state automata. Numerous studies have underscored the adequacy of automata for linguistic problems at all descriptive levels from morphology and syntax to phonetic issues. The grammars created with Unitex carry this approach further by using a formalism even more powerful than automata. These grammars are represented as graphs that the user can easily create and update.

The tables built in the context of lexicon-grammar are matrices describing properties of certain words. Many such tables have been constructed for all simple verbs in French as a way of describing their relevant properties. Experience has shown that every word has a quasi-unique behavior, and these tables are a way of presenting the grammar of every element in the lexicon, hence the name lexicon-grammar for this linguistic theory. Unitex offers a way to directly construct grammars from these tables.

Unitex can be viewed as tool with which one can put these linguistic resources to use. Its technical characteristics are its portability, modularity, the possibility of dealing with languages that use special writing systems (e.g. many Asian languages), and its openness, thanks to its open source distribution. Its linguistic characteristics are the ones that have motivated the elaboration of these resources: the precision, the completeness, and the taking into account of frozen expressions, most notably those which concern the enumeration of compound words.

The first chapter describes how to install and run Unitex.

Chapter 2 presents the different steps in the analysis of a text.

Chapter 3 describes the formalism of the DELA electronic dictionaries and the different operations that can be applied to them.

Chapters 4 and 5 present different means for making text searches more effective. Chapter 5 describes in detail how the graph editor is used.

Chapter 6 is concerned with the different possible applications of grammars. The particularities of each type of grammar are presented.

Chapter 7 introduces the concept of a text automaton and describes the properties of this notion. This chapter also describes the operations on this object, in particular, how to disambiguate lexical items with the ELAG program.

Chapter 8 contains an introduction to lexical-grammar tables, followed by a description of the method of constructing grammars based on these tables.

Chapter 9 contains a detailed description of the different external programs that make up the Unitex system.

Chapter 10 contains descriptions of all file formats used in the system.

Chapter 1

Installing Unitex

Unitex is a multi-platform system that runs on Windows as well as on Linux or MacOS. This chapter describes how to install and how to launch Unitex on any of these systems. It also presents the procedures used to add new languages and to uninstall Unitex.

1.1 Licenses

Unitex is a free software. This means that the sources of the programs are distributed with the software, and that anyone can modify and redistribute them. The code of the Unitex programs is under the LGPL licence ([**?**]), except for the TRE library for dealing with regular expressions from Ville Laurikari ([**?**]), which is under GPL licence ([**?**]). The LGPL Licence is more permissive than the GPL licence, because it makes it possible to use LGPL code in nonfree software. From the point of view of the user, there is no difference, because in both cases, the software can freely be used and distributed.

1.2 The Java runtime environment

Unitex consists of a graphical interface written in Java and external programs written in C/C++. This mixture of programming languages is responsible for a fast and portable application that runs on different operating systems. Before you can use the graphical interface you first have to install the runtime environment, usually called or ().

For the graphical mode, Unitex needs Java version 1.4 (or later). If you have an older version of Java, Unitex will stop once you have selected the working language. You can download the virtual machine for your operating system for free from the Sun Microsystems web site at the following address: `http://java.sun.com` If you're working on Linux, or if you are using a Windows version with personal user accounts you will have to ask your system administrator to install Java.

1.3 Installation on Windows

If Unitex is to be installed on a multi-user Windows machine, it is recommended that the systems administrator performs the installation. If you are the only user on your machine, you can perform the installation yourself.

Decompress the file `unitex_1.2.zip` (You can download this file from the following address: `http://www-igm.univ-mlv.fr/~unitex`) into a directory `Unitex` that should preferably be created within the `Program Files` folder. After decompressing the file, the `Unitex` directory contains several subdirectories one of which is called `App`. This directory contains a file called `Unitex.jar`. This file is the Java executable that launches the graphical interface. You can double click on this icon to start the program. To facilitate launching Unitex, you may want to add a shortcut to this file on the desktop.

1.4 Installation on Linux and Mac OS X

In order to install Unitex on Linux, it is recommended to have system administrator permissions. Decompress the file `unitex_1.2.zip` in a directory named `Unitex`, by using the following command:

```
unzip Unitex_1.2.zip -d Unitex
```

Within the directory `Unitex/Src/C++`, start the compilation of Unitex with the command:

```
make install
```

You can then create an alias in the following way:

```
alias unitex='cd /..../Unitex/App/ ; java -jar Unitex.jar'
```

1.5 Using Unitex the first time

If you working with Windows, the program will ask you to choose a personal working directory, which you can change later. To create a directory, click on the icon showing a file (see figure 1.3).

If your are using Linux, the program will automatically create a `/unitex` directory in your `$HOME` directory. This directory allows you to save your personal data. For each language that you will be using, the program will copy a root directory of that language to your peronal work directory, except the dictionaries. You can also modify your copy of the files without risking to damage the system files.

1.6 Adding new languages

There are two different ways to add languages. If you want to add a language that is to be accessible by all users, you have to copy the corresponding directory to the `Unitex` system directory, for which you will need to have the access rights (this might mean that you need

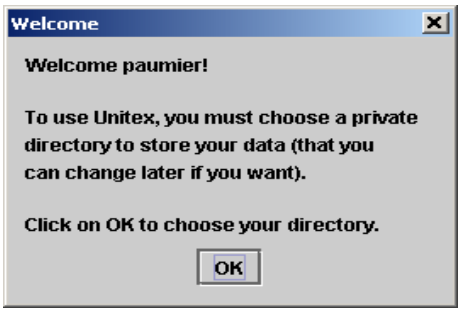

Figure 1.1: First usage on Windows

Figure 1.2: First usage on Linux

to ask your system administrator to do it). On the other hand, if the language is only used by a single user, he can also copy the directory to his working directory. He can work with this language without this language being shown to other users.

1.7 Uninstalling Unitex

No matter which operating system you are working with, it is sufficient to delete the `Unitex` directory to completely delete all the program files. Under Windows you may have to delete the shortcut to `Unitex.jar` if you have created one on your desktop. The same has to be done on Linux, if you have created an alias.

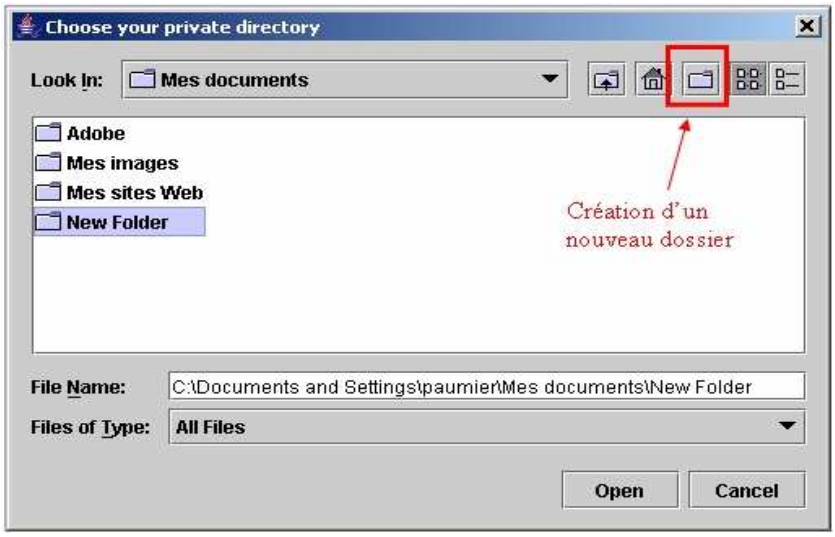

Figure 1.3: Creating the personal work directory

Chapter 2

Loading a text

One of the main functionalities of Unitex is being able to search for expressions within a text. For this, texts have to undergo a set of preprocessing steps that normalize non-ambiguous forms and split the text in sentences. Once these operations are performed, the electronic dictionaries are applied to the texts. Then one can search more effectively in the texts by using grammars.

This chapter describes the different steps for text preprocessing.

2.1 Selecting a language

When starting Unitex, the program asks you to choose the language in which you want to work (see figure 2.1). The languages displayed are the ones that are present in the system directory `Unitex` and those that are installed in your personal working directory. If you use a language for the first time, Unitex copies the system directory for this language to your personal directory, except for the dictionaries.

Choosing the language allows Unitex to find certain files, for example the alphabet file. You can change the language at any time by choosing "Change Language..." in the "Text" menu. If you change the language, the program will close all windows related to the current text, if there are any. The active language is indicated in the title bar of the graphical interface.

2.2 Text formats

Unitex works with Unicode texts. Unicode is a standard that describes a universal character code. Each character is given a unique number which allows to represent texts without having to take into account the proprietary codes on different machines and/or operating systems. Unitex uses a two-byte representation of the Unicode 3.0 standard, called Unicode Little-Endian (for more details, see [**?**]).

Figure 2.1: Language selection when starting Unitex

The texts that come with Unitex are already in Unicode format. If you try to open a text that is not in the Unicode format, the program proposes to convert it automatically (see figure 2.2).

This conversion is based on the current language: if you are working in French, Unitex proposes to convert your text[1] assuming that it is coded using a French code page. By default, Unitex proposes to either replace the original text or to rename the original file by inserting `.old` at the beginning of its extension. For example, if one has an ASCII file named `balzac.txt`, the conversion process will create a copy of this ASCII file named `balzac.old.txt`, and will replace the contents of `balzac.txt` with its equivalent in Unicode.

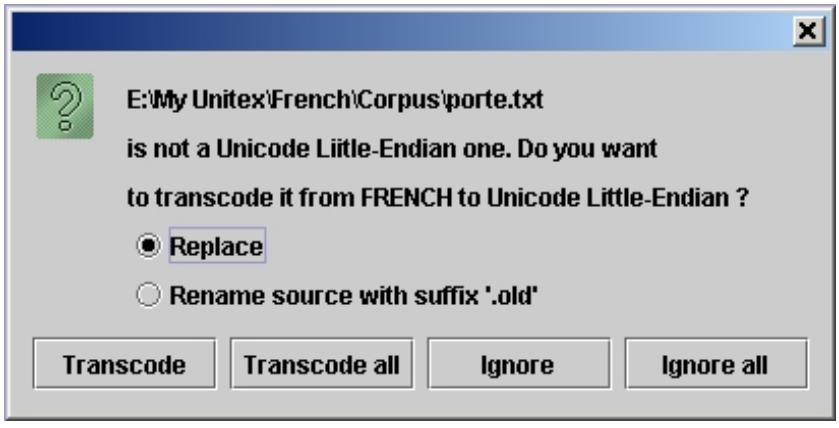

Figure 2.2: Automatic conversion of a non-Unicode text

If the encoding suggested by default is not correct or if you want to rename the file differently than with the suffix `.old`, you can use the "Transcode Files" command in the "File Edition" menu. This command enables you to choose source and target encodings of

[1] Unitex also proposes to automatically convert graphs and dictionaries that are not in Unicode Little-Endian.

the documents to be converted (see figure 2.3). By default, the proposed source encoding is that which corresponds to the current language and the destination encoding is Unicode Little-Endian. You can modify these choices by selecting any source and target encoding. Thus, if you wish, you can convert your data into other encodings, as for example UTF-8 in order for instance to create web pages. The button "Add Files" enables you to select the files to be converted. The button "Remove Files" makes it possible to remove a list of files erroneously selected. The button "Transcode" will start the conversion of all the selected files. If an error occurs when the file is processed (for example, a file which is already in Unicode), the conversion continues with the next file.

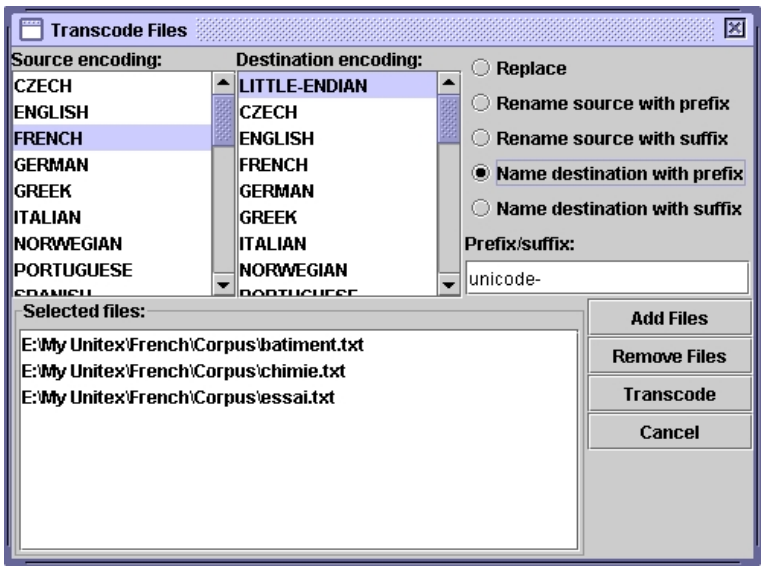

Figure 2.3: File conversion

To obtain a text in the right format, you can also use a text processor like the free software from OpenOffice.org ([?]) or Microsoft Word, and save your document with the format "Unicode text".

By default, the encoding proposed on a PC is always Unicode Little-Endian. The texts thus obtained do not contain any formatting information anymore (fonts, colors , etc.) and are ready to be used with Unitex.

2.3 Editing texts

You also have the possibility of using the text editor integrated into Unitex, accessible via the "Open..." command in the "File Edition" menu". This editor offers search and replace functionalities for the texts and dictionaries handled by Unitex. To use it, click on the "Find" icon. You will then see a window divided into three parts. The "Find" part corresponds to the usual search operations. If you open a text split into sentences, you will have the possibility to search by the number of a sentence in the "Find Sentence" part. Lastly, the "Search

Figure 2.4: Saving in Unicode with Microsoft Word

Dictionary" part, visible in figure 2.5, enables you to carry out operations concerning the electronic dictionaries. In particular, you can search by specifying if it concerns inflected terms, lemmas, the grammatical and semantic and/or the inflecional codes. Thus, if you want to search for all the verbs which have the semantic feature `t`, which indicates transitivity, it is enough to search for `t` by clicking on "Grammatical code". You will get the matching entries without confusion with all the other occurrences of the letter `t`.

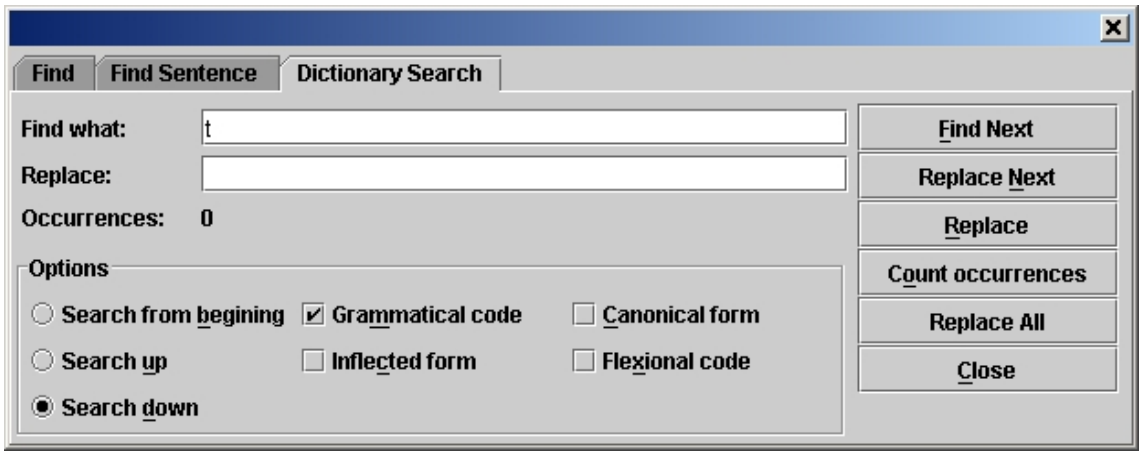

Figure 2.5: Searching for the semantic feature `t` in an electronic dictionary

2.4 Opening a text

Unitex deals with two types of text files.

The files with the extension `.snt` are text files preprocessed by Unitex which are ready to be manipulated by the different system functions. The files ending with `.txt` are raw files.

To use a text, open the `.txt` file by clicking on "Open..." in the "Text" menu.

Figure 2.6: Text Menu

Choose the file type "Raw Unicode Texts" and select your text:

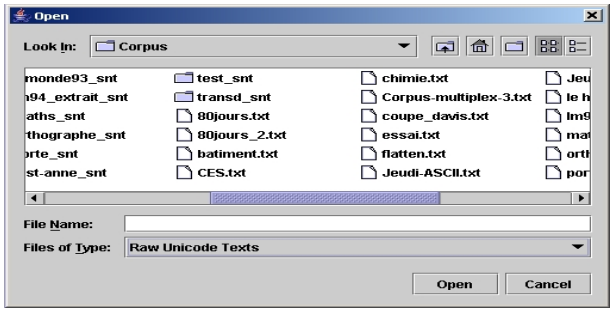

Figure 2.7: Opening a Unicode text

Files larger than 5 MBytes are not displayed. The message "`This file is too large to be displayed. Use a word processor to view it.`" is displayed in the window. This limit applies to all open text files (the list of lexical units, dictionaries, etc.). To modify this limit, use the menu "Info>Preferences" and set the new value for "Maximum Text File Size" in the tab "Text Presentation" (see 4.7, page 53).

2.5 Preprocessing a text

After a text is selected, Unitex offers to preprocess it. Text preprocessing consists of performing the following operations: Normalization of separators, identification of lexical units, normalization of non-ambiguous forms, splitting into sentences and the application of dictionaries. If you choose not to preprocess the text, it will nevertheless be normalized and lexical units will be looked up, since these operations are necessary for all further Unitex operations. It is always possible to carry out the preprocessing later by clicking on "Preprocess Text..." in the "Text" menu.

If you choose to preprocess the text, Unitex proposes to parameterize it as in the window shown in figure 2.8.

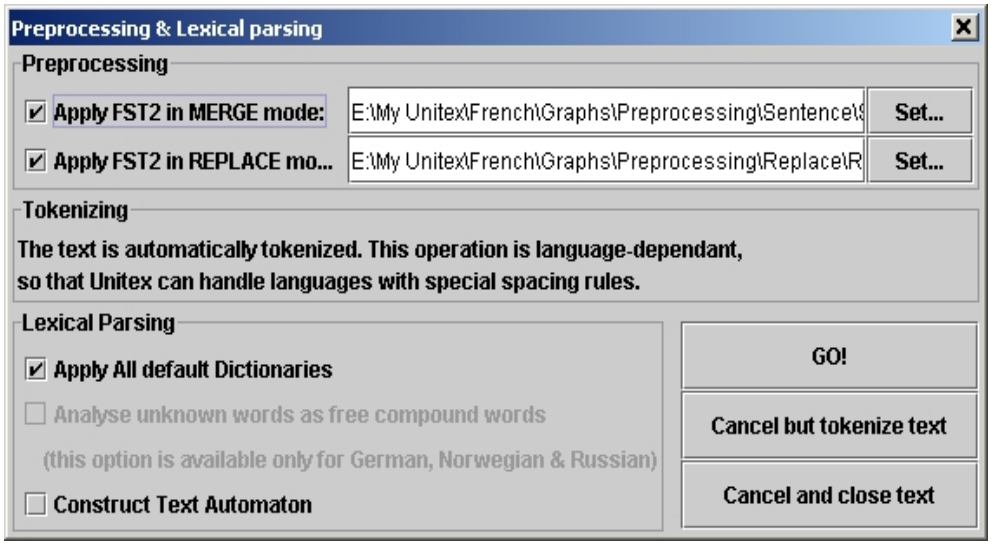

Figure 2.8: Preprocessing Window

The option "Apply FST2 in MERGE mode" is used to split the text into sentences. The option "Apply FST2 in REPLACE mode" is used to make replacements in the text, especially for the normalization of non-ambiguous forms. With the option "Apply All default Dictionaries" you can apply dictionaries in the DELA format (Dictionnaires Electroniques du LADL). The option "Analyze unknown words as free compound words" is used in Norwegian for correctly analyzing compound words constructed via concatenation of simple forms. Finally, the option "Construct Text Automaton" is used to build the text automaton. This option is deactivated by default, because it consumes a large amount of memory and disk space if the text is too large. The construction of the text automaton is described in chapter 7.

NOTE: If you click on "Cancel but tokenize text", the program will carry out the normalization of separators and look up the lexical units. Click on "Cancel and close text" to cancel the operation.

2.5.1 Normalization of separators

The standard separators are the space, the tab and the newline characters. There can be several separators following each other, but since this isn't useful for linguistic analyses, separators are normalized according to the following rules:

- separators that contain at least one newline are replaced by a single newline
- all other sequences of separators are replaced by a space.

The distinction between space and newline is maintained at this point because the presence of newlines may have an effect on the process of splitting the text into sentences. The result of the normalization of a text named `my_text.txt` is a file in the same directory as the `.txt` file and is named `my_text.snt`.

NOTE: When the text is preprocessed using the graphical interface, a directory named `my_text_snt` is created immediately after normalization. This directory, called text directory, contains all the data associated with this text.

2.5.2 Splitting into sentences

Splitting texts into sentences is an important preprocessing step since this helps in determining the units for linguistic processing. The splitting is used by the text automaton construction program. In contrast to what one might think, detecting sentence boundaries is not a trivial problem. Consider the following text:

The family urgently called Dr. Martin.

The full stop that follows *Dr* is followed by a word beginning with a capital letter. Thus it may be considered as the end of the sentence, which would be wrong. To avoid the kind of problems caused by the ambiguous use of punctuation, grammars are used to describe the different contexts for the end of a sentence. Figure 2.9 shows an example grammar for sentence splitting.

When a path of the grammar recognizes a sequence in the text and when this path produces the sentence separator symbol `{S}` , this symbol is inserted into the text.

The path shown at the top of figure 2.9 recognizes the sequence consisting of a question mark and a word beginning with a capital letter and inserts the symbol `{S}` between the question mark and the following word. The following text:

What time is it? Eight o' clock.

will be converted to:

What time is it ?{S} Eight o' clock.

A grammar for splitting can use the following special symbols:

- <E> : empty word, or epsilon. Recognizes the empty sequence;
- <MOT> : recognizes any sequence of letters;
- <MIN> : recognizes any sequence of letters in lower case;
- <MAJ> : recognizes any sequence of letters in upper case
- <PRE> : recognizes any sequence of letters that begins with an upper case letter
- <NB> : recognizes any sequence of digits (1234 is recognized but not 1 234);
- <PNC> : recognizes the punctuation symbols ; , ! ? : and the inverted exclamation points and question marks in Spanish and some Asian punctuation letters
- <^> : recognizes a newline;
- # : prohibits the presence of a space.

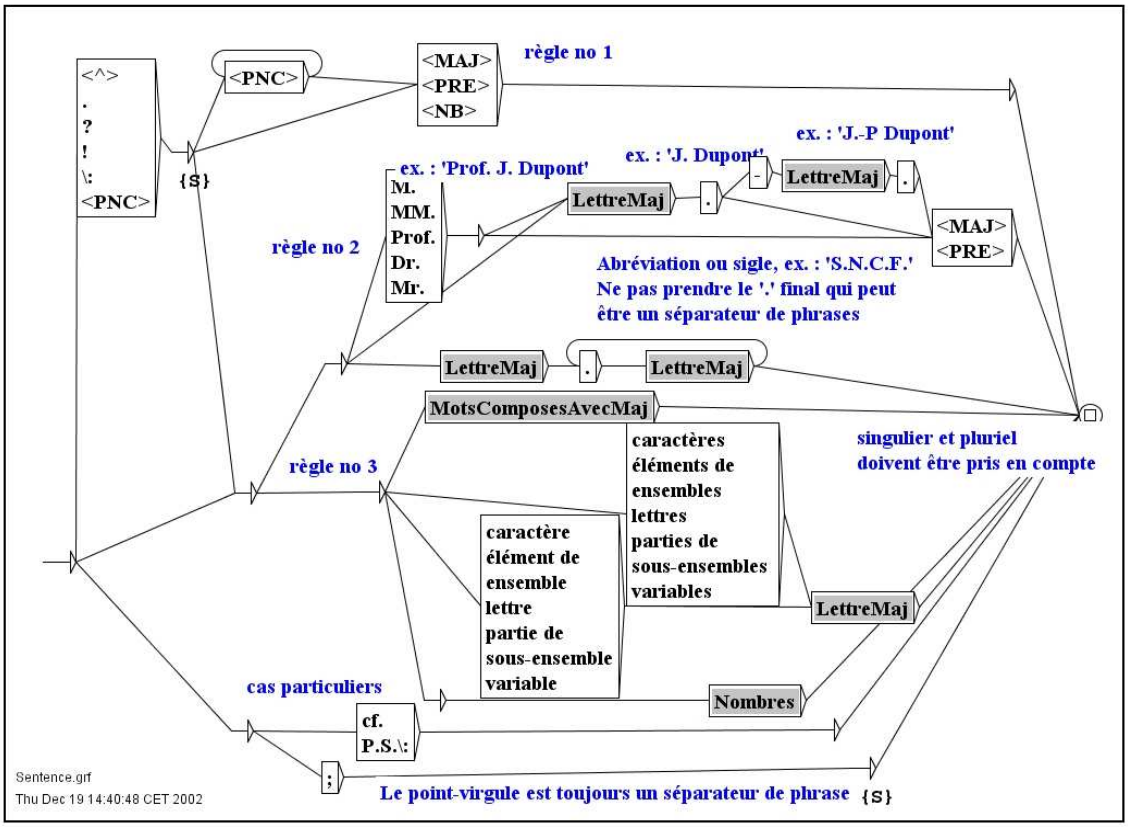

Figure 2.9: Sentence splitting grammar for French

By default, the space is optional between two boxes. If you want to prohibit the presence of this separator you have to use the special separator #. Lower and upper case letters

are defined by an alphabet file (see chapter 10). For more details on grammars, see chapter 5. The grammar used here is named `Sentence.fst2` and can be found in the following directory:

```
/(user home directory)/(language)/Graphs/Preprocessing/Sentence
```

This grammar is applied to a text with the `Fst2Txt` program in MERGE mode. This has the effect that the output produced by the grammar, in this case the symbol `{S}`, is inserted into the text. This program takes a `.snt` file and modifies it.

2.5.3 Normalization of non-ambiguous forms

Certain forms present in texts can be normalized (for example, the English sequence "*I'm*" is equivalent to "*I am*"). You may want to replace these forms according to your own needs. However, you have to be careful that the forms normalized are unambiguous or that the removal of ambiguities has no undesirable consequences.

For instance, if you want to normalize "*O'clock*" to "*of the clock*", it would be a bad idea to replace "*O'*" by "*of the* ", because a sentence like:

John O'Connor said: "it's 8 O'clock"

would be replaced by the following incorrect sentence:

John of the Connor said: "it's 8 of the clock"

Thus, one needs to be very careful when using the normalization grammar.

One needs to pay attention to spaces as well. For example, if one replaces "'re" by "are", the sentence:

You're stronger than him.

will be replaced by:

You are stronger than him.

To avoid this problem, one should explicitly insert a space, i.e. replace "*'re*" by " *are*".

The accepted symbols for the normalization grammar are the same as the ones allowed for the sentence splitting grammar. The normalization grammar is called Replace.fst2 and can be found in the following directory:

```
/(home directory)/(active language)/Graphs/Preprocessing/Replace
```

As in the case of sentence splitting, this grammar is applied using the Fst2Txt program, but in REPLACE mode, which means that input sequences recognized by the grammar are replaced by the output sequences that are produced. Figure 2.10 shows a grammar that normalizes verbal contractions in French.

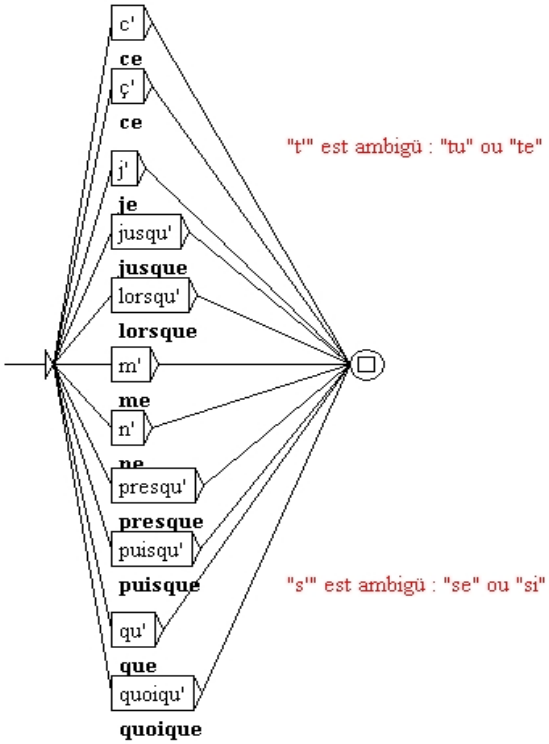

Figure 2.10: Normalization grammar for some elisions in French

2.5.4 Splitting a text into lexical units

Some languages, in particular Asian languages, use separators that are different from the ones used in western languages. Spaces can be forbidden, optional, or mandatory. In order to better cope with these particularities, Unitex splits texts in a language dependent way.

Thus, languages like French are treated as follows:

A lexical unit can be:

- the phrase separator `{S}`;
- a lexical tag `{aujourd'hui,.ADV}`;
- a sequence of letters (the letters are defined in the language alphabet file);
- a non-word character; if it is a newline, it is replaced by a space.

For other languages, splitting is done on a character by character basis, except for the phrase separator `{S}` and lexical tags.

This simple splitting is fundamental for the use of Unitex, but limits the optimization of search operations for patterns.

Regardless of the mechanism used, the newlines in a text are replaced by spaces.

Splitting is done by the `Tokenize` program. This program creates several files that are saved in the text directory:

- `tokens.txt` contains the list of lexical units in the order in which they are found in the text;
- `text.cod` contains the position table; every number in this table corresponds to the index of a lexical unit in the file `tokens.txt`;
- `tok_by_freq.txt` contains the list of lexical units sorted by frequency;
- `tok_by_alph.txt` contains the list of lexical units in alphabetical order;
- `stats.n` contains some statistics about the text.

Splitting the text:

A cat is a cat.

returns the following list of lexical units: *A* SPACE *cat is a* .

You will observe that tokenization is case sensitive (A and a are two distinct tokens), and that each token is listed only once. Numbering these tokens with 0 to 5, the text can be represented by a sequence of numbers as described in the following table:

Indice	0	1	2	1	3	1	4	1	2	5
Corresponding lexical unit	*A*		*cat*		*is*		*a*		*cat*	.

Table 2.1: Representing the text *A cat is a cat.*

For more details, see chapter 10.

2.5.5 Applying dictionaries

Applying dictionaries consists of building the subset of dictionaries consisting only of forms that are present in the text. Thus, the result of applying a English dictionary to the text *Igor's father in law is ill* produces a dictionary of the following simple words:

```
father,.N+Hum:s
father,.V:W:P1s:P2s:P1p:P2p:P3p
ill,.A
ill,.ADV
ill,.N:s
```

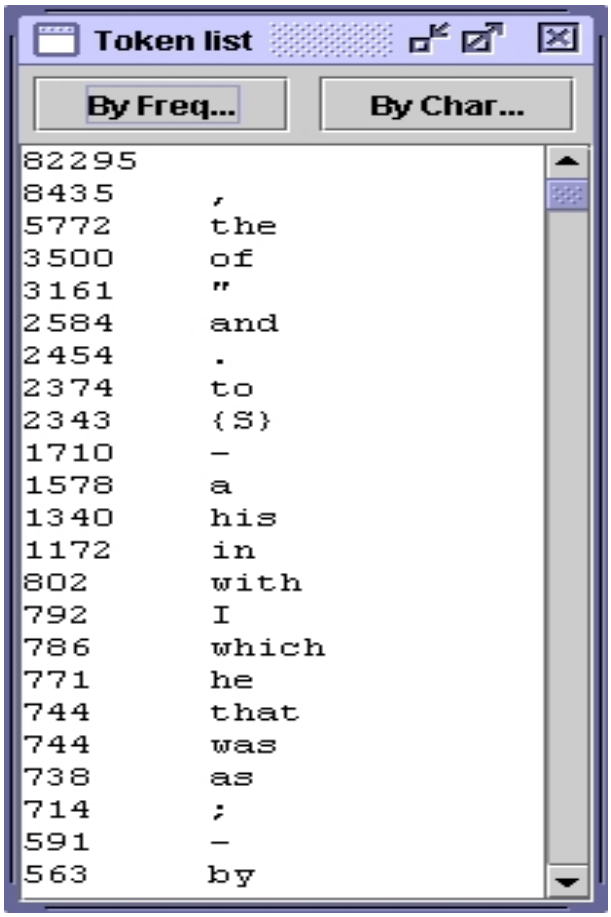

Figure 2.11: Lexical units in an English text sorted by frequency

```
in,.A
in,.N:s
in,.PART
in,.PREP
is,be.V:P3s
is,i.N:p
law,.N:s
law,.V:W:P1s:P2s:P1p:P2p:P3p
s,.N:s
```

as well as a dictionary of compound words consisting of a single entry:

```
father in law,.N+NPN+Hum+z1:s
```

Since the sequence *Igor* is neither a simple English word nor a part of a compound word, it is treated as an unknown word. The application of dictionaries is done through the program `Dico`. The three files produced (`dlf` for simple words, `dlc` for compound words and

`err` for unknown words) are placed in the text dictionary. The `dlf` and `dlc` files are called text dictionaries.

As soon as the dictionary look-up is finished, Unitex displays the sorted lists of simple, compound and unknown words found in a new window. Figure 2.12 shows the result for an English text.

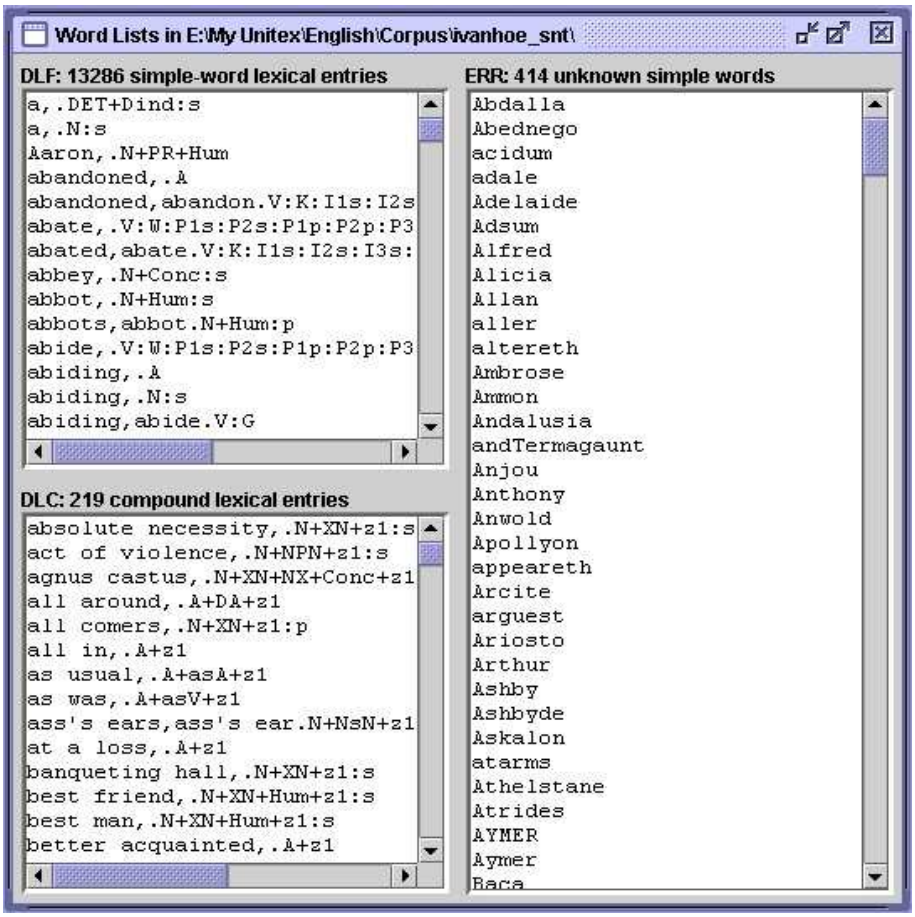

Figure 2.12: Result after applying dictionaries to an English text

It is also possible to apply dictionaries without preprocessing the text. In order to do this, click on "Apply Lexical Resources..." in the "Text" menu. Unitex then opens a window (see figure 2.13) in which you can select the list of dictionaries to apply.

The list "User resources" lists all compressed dictionaries present in the directory (`current language` of the user. The dictionaries installed in the system are listed in the scroll list named "System resources". Use the combination<Ctrl+click> to select multiple dictionaries. The button "Set Default" allows you to define the current selection of dictionaries as the default. This default selection will then be used during preprocessing if you activate the option "Apply All default Dictionaries".

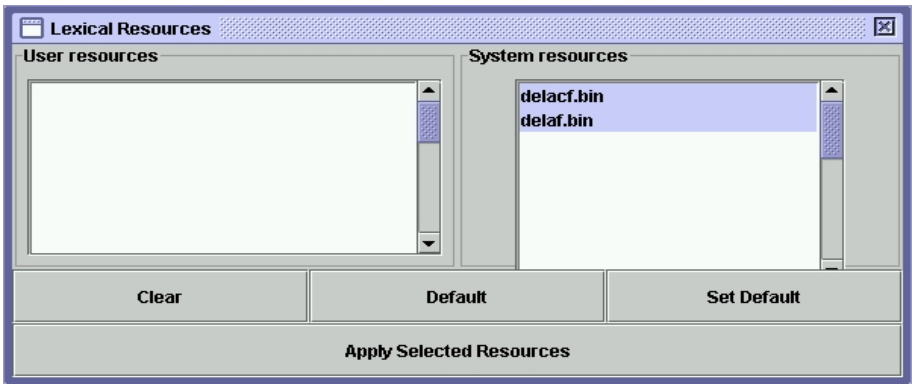

Figure 2.13: Parameterizing the application of dictionaries

2.5.6 Analysis of compound words in German, Norwegian and Russian

In certain languages like Norwegian, German and others, it is possible to form new compound words by concatenating together other words. For example, the word *aftenblad* meaning *evening journal* is obtained by combining the words *aften* (*evening*) et *blad* (*journal*). The program `PolyLex` [**?**] searches the list of unknown words after the application of dictionaries and tries to treat each of these words as a compound word. If a word has at least one analysis as a compound word, it is deleted from the list of unknown words and the lines produced for this word are appended to the text dictionary of simple words.

Chapter 3

Dictionaries

3.1 The DELA dictionaries

The electronic dictionaries distributed with Unitex use the DELA syntax (Dictionnaires Electroniques du LADL, LADL electronic dictionaries). This syntax describes the simple and compound lexical entries of a language with their grammatical, semantic and inflectional information. We distinguish two kinds of electronic dictionaries. The one that is used most often is the dictionary of inflected forms DELAF (DELA de formes Fléchies, DELA of inflected forms) or DELACF (DELA de formes Composées Fléchies, DELA of compound inflected forms) in the case of compound forms. The second type is a dictionary of non-inflected forms called DELAS (DELA de formes simples, simple forms DELA) or DELAC (DELA de formes composées, compound forms DELA).

Unitex programs make no distinction between simple and compound form dictionaries. We will use the terms DELAF and DELAS to distinguish the two kinds of dictionaries whose entries are simple, compound, or mixed forms.

3.1.1 The DELAF format

Entry syntax

An entry of a DELAF is a line of text terminated by a newline that conforms to the following syntax:

```
apples,apple.N+conc:p/this is an example
```

The different elements of this line are:

- `apples` is the inflected form of the entry; it is mandatory;
- `apple` is the canonical form of the entry. For nouns and adjectives (in French), it is usually the masculine singular form; for verbs, it is the infinitive. This information may be left out as in the following example:

```
apple,.N+conc:s
```

This means that the canonical form is the same as the inflected form. The canonical form is separated from the inflected form by a comma.

- `N+conc` is the sequence of grammatical and semantic information. In our example, `N` designates a noun, and `conc` indicates that this noun designates a concrete object (see table 3.2).

  Each entry must have at least one grammatical or semantic code, separated from the canonical form by a period. If there are more codes, these are separated by the `+` character.

- `:p` is an inflectional code which indicates that the noun is plural. Inflectional codes are used to describe gender, number, declination, and conjugation. This information is optional. An inflectional code is made up of one or more characters that represent one information each. Inflectional codes have to be separated by the `:` character, for instance in an entry like the following:

  ```
  hang,.V:W:P1s:P2s:P1p:P2p:P3p
  ```

  The `:` character is interpreted in OR semantics. Thus, `:W:P1s:P2s:P1p:P2p:P3p` means "infinitive", "1st person singular present", "2nd person singular present", etc. (see table 3.3) Since each character represents one information, it is not necessary to use the same character more than once. In this way, encoding the past participle using the code `:PP` would be exactly equivalent to using `:P` alone;

- `/this is an example` is a comment. Comments are optional and may be introduced by the `/` character. These comments are left out when the dictionaries are compressed.

IMPORTANT REMARK: It is possible to use the full stop and the comma within a dictionary entry. In order to do this they have to be escaped using the `\` character:

```
1\,000,one thousand.NUMBER
United Nations,U\.N\..ACRONYM
```

ATTENTION: Each character is taken into account within a dictionary line. For example, if you insert spaces, they are considered to be a part of the information. In the following line:

```
hath,have.V:P3s /old form of 'has'
```

The space that precedes the `/` character will be considered to be one of the 4 inflectional codes `P`, `3`, `s` and space.

It is possible to insert comments into a DELAF or DELAS dictionary by starting the line with a / character. Example:

```
/ in the next entry the backslash escapes the comma:
1\,000,one thousand.NUMBER
```

Compound words with spaces or dashes

Certain compound words like *acorn-shell* can be written using spaces or dashes. In order to avoid duplicating the entries, it is possible to use the = character. At the time when the dictionary is compressed, the `Compress` program verifies for each line if the inflected or canonical form contains a non-escaped = character. If this is the case, the programm replaces this by two entries: The one where the = character is replaced by a space, and one where it is replaced by a dash. Thus, the following entry:

```
acorn=shells,acorn=shell.N:p
```

is replaced by the following entries:

```
acorn shells,acorn shell.N:p
acorn-shells,acorn-shell.N:p
```

NOTE: If you want to keep an entry that includes the = character, escape it using \ like in the following example:

```
E\=mc2,.FORMULA
```

This replacement is done when the dictionary is compressed. In the compressed dictionary, the escaped = characters are replaced by simple =. As such, if a dictionary containing the following lines is compressed:

```
E\=mc2,.FORMULA

acorn=shell,.N:s
```

and if the dictionary is applied to the following text:

Formulas like E=mc2 have nothing to do with acorn-shells.

you will get the following lines in the dictionary of compound words of the text:

```
E=mc2,.FORMULA

acorn-shells,.N:p
```

Entry Factorization

Several entries containing the same inflectional and canonical forms can be combined into a single one if they have the same grammatical and semantic codes. Among other things this allows us to combine identical conjugations for a verb:

```
bottle,.V:W:P1s:P2s:P1p:P2p:P3p
```

If the grammatical and semantic information differ, one has to create distinct entries:

```
bottle,.N+Conc:s
bottle,.V:W:P1s:P2s:P1p:P2p:P3p
```

Certain entries that have the same grammatical and semantic entries can have different senses, as it is the case for the French word *poêle* that describes a stove or a net in the masculine sense and a kitchen instrument in the feminine sense. You can thus distinguish the entries in this case:

```
poêle,.N+z1:fs/ poêle à frire
poêle,.N+z1:ms/ voile, linceul; appareil de chauffage
```

NOTE: In practice this distinction has the only consequence that the number of entries in the dictionary increases.

In the different programs that make up Unitex these entries are reduced to

```
poêle,.N+z1:fs:ms
```

Whether this distinction is made is thus left to the maintainers of the dictionaries.

3.1.2 The DELAS Format

The DELAS format is very similar to the one used in the DELAF. The only difference is that there is only one canonical form followed by grammatical and/or semantic codes. The canonical form is separated from the different codes by a comma. There is an example:

```
horse,N4+Anl
```

The first grammatical or semantic code will be interpreted by the inflection program as the name of the grammar used to inflect the entry. The entry of the example above indicates that the word *horse* has to be inflected using the grammar named `N4`. It is possible to add inflectional codes to the entries, but the nature of the inflection operation limits the usefulness of this possibility. For more details see below in section 3.4.

3.1.3 Dictionary Contents

The dictionaries provided with Unitex contain descriptions of simple and compound words. These descriptions indicate the grammatical category of each entry, optionally their inflectional codes, and diverse semantic information. The following tables give an overview of some of the different codes used in the Unitex dictionaries. These codes are the same for almost all languages, though some of them are special for certain languages (*i.e.* code for neuter nouns, etc.).

Code	Description	Examples
A	adjective	fabulous, broken-down
ADV	adverb	actually, years ago
CONJC	coordinating conjunction	but
CONJS	subordinating conjunction	because
DET	determiner	each
INTJ	interjection	eureka
N	noun	evidence, group theory
PREP	preposition	without
PRO	pronoun	you
V	verb	overeat, plug-and-play

Table 3.1: Frequent grammatical codes

Code	Description	Example
z1	general language	joke
z2	specialized language	floppy disk
z3	very specialized language	serialization
Abst	abstract	patricide
Anl	animal	horse
AnlColl	collective animal	flock
Conc	concrete	chair
ConcColl	collective concrete	rubble
Hum	human	teacher
HumColl	collective human	parliament
t	transitive verb	kill
i	intransitive verb	agree

Table 3.2: Some semantic codes

NOTE: The descriptions of tense in table 3.3 correspond to French. Nontheless, the majority of these definitions can be found in other languages (infinitive, present, past participle, etc.).

In spite of a common base in the majority of languages, the dictionaries contain encoding particularities that are specific for each language. Thus, as the declination codes vary a lot between different languages, they are not described here. For a complete description of all codes used within a dictionary, we recommend that you contact the author of the dictionary directly.

However, these codes are not exclusive. A user can introduce codes himself and can create his own dictionaries. For example, for educational purposes one could use a marker "faux-ami" in an English dictionary:

Code	Description
m	masculine
f	feminin
n	neuter
s	singular
p	plural
1, 2, 3	1st, 2nd, 3rd person
P	present indicative
I	imperfect indicative
S	present subjunctive
T	imperfect subjunctive
Y	present imperative
C	present conditionnal
J	passé simple
W	infinitive
G	present participle
K	past participle
F	future

Table 3.3: Common inflectional codes

```
bless,.V+faux-ami/bénir
cask,.N+faux-ami/tonneau
journey,.N+faux-ami/voyage
```

It is equally possible to use dictionaries to add extra information. Thus, you can use the inflected form of an entry to describe an abbreviation and the canonical form to provide the complete form:

```
DNA,DeoxyriboNucleic Acid.ACRONYM
LADL,Laboratoire d'Automatique Documentaire et Linguistique.ACRONYM
UN,United Nations.ACRONYM
```

3.2 Verfication of the dictionary format

When dictionaries become larger, it becomes tiresome to verify them by hand. Unitex contains the program `CheckDic` that automatically verifies the format of DELAF and DELAS dictionaries.

This program verifies the syntax of the entries. For each malformed entry the program outputs the line number, the contents of the line and the type of error. The results are saved in the file `CHECK_DIC.TXT` which is displayed when the verification is finished. In addition to eventual error messages, the file also contains the list of all characters used in the inflectional and canonical forms, the list of grammatical and semantic codes, and the list

of inflectional codes used. The character list makes it possible to verify that the characters used in the dictionary are consistent with those in the alphabet file of the language. Each character is followed by its value in hexadecimal notation.

These code lists can be used to verify that there are no typing errors in the codes of the dictionary.

The program works with non-compressed dictionaries, i.e. the files in text format. The general convention is to use the `.dic` extension for these dictionaries. In order to verify the format of a dictionary, you first open it by choosing "Open..." in the "DELA" menu.

Figure 3.1: "DELA" Menu

Let's load the dictionary as in figure 3.2:

Figure 3.2: Dictionary example

In order to start the automatic verification, click on "Check Format..." in the "DELA" menu. A window like in figure 3.3 is opened:

Figure 3.3: Automatic verification of a dictionary

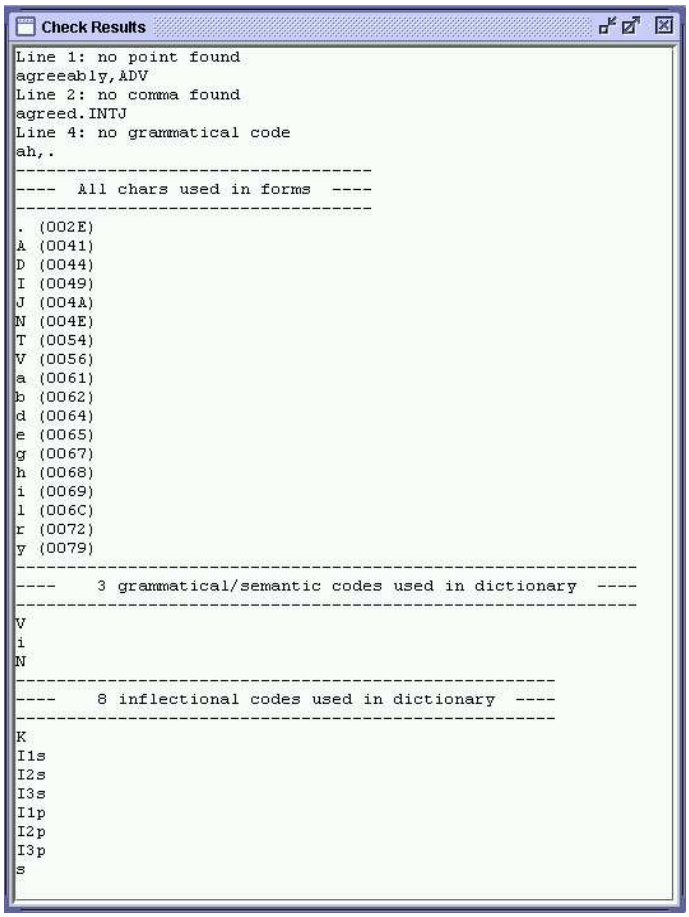

Figure 3.4: Results of the automatic verification

In this window you choose the dictionary type you want to verify. The results of verifying the dictionary in figure 3.2. are shown in figure 3.4.

The first error is caused by a missing period. The second, by the fact that no comma was found after the end of an inflected form. The third error indicates that the program didn't find any grammatical or semantic codes.

3.3 Sorting

Unitex uses the dictionaries without having to worry about the order of the entries. When displaying them it is sometimes preferable to sort the dictionaries. The sorting depends on a number of criteria, first of all on the language of the text. Therefore the sorting of a Thai dictionary is done according to an order different from the alphabetical order. So different in fact that Unitex uses a sorting procedure developed specifically for Thai 9).

For European languages the sorting is usually done in terms of the lexicographical order, although there are some variants. Certain languages like French treat some characters as

equivalent. For example, the difference between the characters `e` and `é` is ignored if one wants to compare the words `manger` et `mangés` because the contexts r and s allow to decide the order. The difference is only taken into account when the contexts are identical, as they are when comparing `pêche` and `pèche`.

To allow for such phenomena, the sort program `SortTxt` uses a file which defines the equivalence of characters. This file is named `Alphabet_sort.txt` and can be found in the user directory for the current language. By default the first lines of this file for French look like this:

```
AÀÂÄaàâä
Bb
CÇcç
Dd
EÉÈÊËeéèêë
```

Characters in the same line are considered equivalent if the context permits. If two equivalent characters must be compared, they are sorted in the order they appear in from left to right. As can be seen from the extract above, there is no difference between lower and upper case. Accents and the cédille character are ignored as well.

To sort a dictionary, open it and then click on "Sort Dictionary" in the DELA menu. By default, the program always looks for the file `Alphabet_sort.txt` . If that file doesn't exist, the sorting is done according to the character indices in the Unicode encoding. By modifying that file, you can define your own sorting order.
Remark: After applying the dictionaries to a text, the files `dlf`, `dlc` and `err` are automatically sorted using this program.

3.4 Automatic inflection

As described in section 3.1.2, a line in a DELAS consists of a canonical form and a sequence of grammatical or semantic codes:

```
aviatrix,N4+Hum
matrix,N4+Math
radix,N4
```

The first code is interpreted as the name of the grammar used to inflect the canonical form. These inflectional grammars have to be compiled (see chapter 5). In the example above, all entries will be inflected by a grammar named `N4`.

In order to inflect a dictionary, click on "Inflect..." in the "DELA" menu. The window in figure 3.5 allows to specify the directory in which inflectional grammars are found. By default, the subdirectory `Inflection` of the directory for the current language is used. The option "Add : before inflectional code if necessary" automatically inserts a ':' character before the inflectional codes if those codes do not start with this character. The option

"Remove class numbers" is used to replace codes with the numbers used in the DELAS by codes without numbers. Example: V17 will be replaced by V.

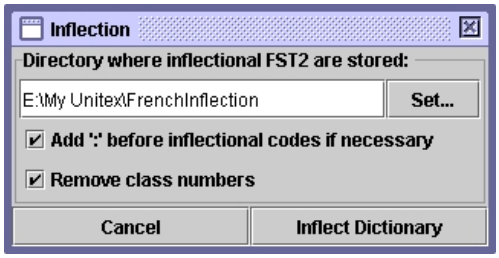

Figure 3.5: Configuration of automatic inflection

Figure 3.6 shows an example of an inflectional grammar:

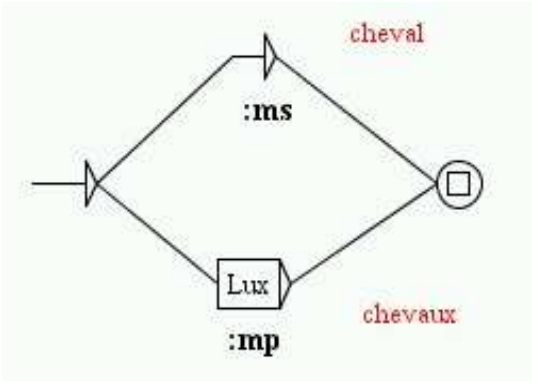

Figure 3.6: Inflectional grammar `N4`

The paths describe the suffixes to add or to remove to get to an inflected form from a canonical form, and the outputs (text in bold under the boxes) are the inflectional codes to add to a dictionary entry.

In our example, two paths are possible. The first doesn't modify the canonical form and adds the inflectional code `:ms`. The second deletes a letter with the `L` operator, then adds the `ux` suffix and adds the inflectional code `:mp`. Three operators are possible: :

- `L` (left) remove a letter from the entry
- `R` (right) restore a letter to the entry. In French, many verbs of the first group are conjugated in the present singular of the third person form by removing the `r` of the infinitive and changing the 4th letter from the end to è: `peler` → `pèle`, `acheter` → `achète`, `gérer` → `gère`, etc. Instead of describing a inflectional suffix for each verb (`LLLLèle`, `LLLLète` et `LLLLère`), the `R` operator can be used to describe it in one step: `LLLLèRR`.

- `C` (copy) duplicates a letter in the entry and moves everything on its right by one position. For example, let us assume that we want to automatically generate the (French) adjectives ending with `able` from the nouns. In cases like `regrettable` or`réquisitionnable`, we see a duplication of the final consonant of the noun. To avoid writing an inflectional graph for every possible final consonant, one can use the `C` operator to duplicate any final consonant.

The inflection program, `Inflect` traverses all paths of the inflectional grammar and tries all possible forms. In order to avoid having to replace the names of inflectional grammars by the real grammatical codes in the dictionary used, the program replaces these names by the longest prefixes made of letters. Thus, `N4` is replaced by `N`. By choosing the inflectional grammar names carefully, one can construct a ready to use dictionary.

Let's have a look at the dictionary we get after the DELAS inflection in our exapmle:

Figure 3.7: Result of automatic inflection

3.5 Compression

Unitex applies compressed dictionaries to the text. The compression reduces the size of the dictionaries and speeds up the lookup.

This operation is done by the `Compress` program. This program takes a dictionary in text form as input (for example `my_dico.dic`) and produces two files:

- `my_dico.bin` contains the minimal automaton of the inflectioned forms of the dictionaries;

- `my_dico.inf` contains the codes that allow to reconstruct the original dictionary from the inflected forms in the `my_dico.bin` file.

The minimal automaton in the `my_dico.bin` file is a representation of inflected forms in which all common prefixes and suffixes are factorized. For example, the minimal automaton of the words `me`, `te`, `se`, `ma`, `ta` et `sa` can be represented by the graph in figure 3.8.

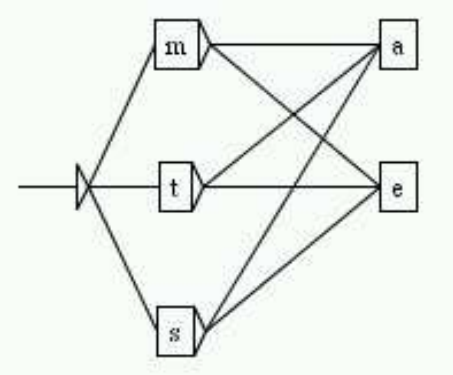

Figure 3.8: Representation of an example of a minimal automaton

To compress a dictionary, open it and click on "Compress into FST" in the "DELA" menu. The compression is independent of the language and of the content of the dictionary. The messages produced by the program are displayed in a window that is not closed automatically. You can see the size of the resulting `.bin` file, the number of lines read and the number of inflectional codes created. Figure 3.9 shows the result of the compression of a dictionary of simple words.

The resulting files are compressed to about 95% for dictionaries containing simple words and 50% for those with compound words.

3.6 Applying dictionaries

Dictionaries can be applied after pre-processing or by explicitly clicking on "Apply Lexical Resources" in the "Text" menu (see section 3.6).

We will now describe the rules for applying dictionaries in detail.

3.6.1 Priorities

The priority rule says that if a word in a text is found in a dictionary, this word will not be taken into account by dictionaries with lower priority.

This allows for eliminating certain ambiguities when applying dictionaries. For example, the word *par* has a nominal interpretation in the golf domain. If you don't want to use this reading, it is sufficient to create a filter dictionary containing only the entry `par,.PREP` and to apply this with highest priority. This way, even if dictionaries of simple words contain a different entry, this will be ignored given the priority rule.

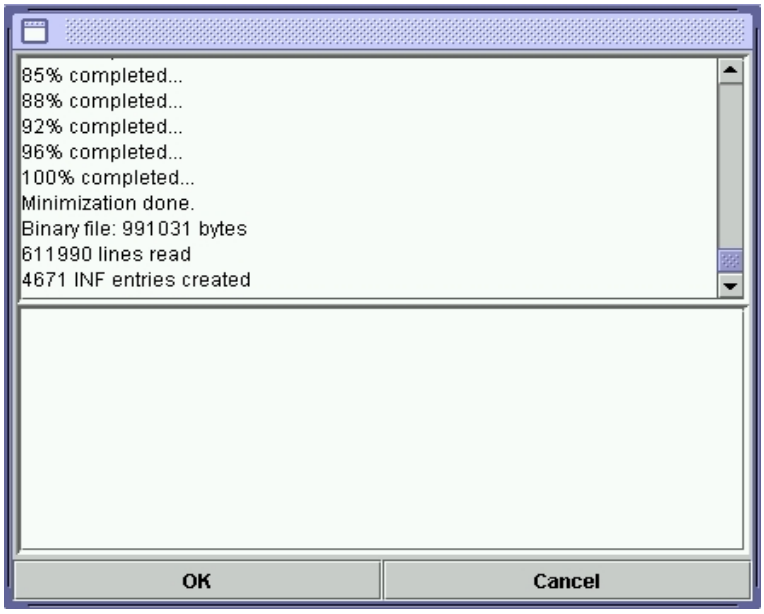

Figure 3.9: Results of a compression

There are three priority levels. The dictionaries whose names without extension end with `-` have the highest priority; those that end with `+` have the lowest one. All other dictionaries are applied with medium priority. The order in which dictionaries with the same priority are applied is not defined. On the command line, the command

```
Dico ex.snt alph.txt countries+.bin cities-.bin rivers.bin regions-.bin
```

will apply the dictionaries in the following order (`ex.snt` is the text to which the dictionaries are applied, and `alph.txt` is the alphabet file used):

1. `cities-.bin`
2. `regions-.bin`
3. `riverts.bin`
4. `countries+.bin`

3.6.2 Application rules for dictionaries

Besides the priority rule, the application of dictionaries respects upper case letters and spaces. The upper case rule is as follows:

- if there is an upper case letter in the dictionary, then an upper case letter has to be in the text;

- if a lower case letter is in the dictionary, there can be either an upper or lower case letter in the text.

Thus, the entry `peter,.N:fs` will match the words `peter`, `Peter` et `PETER`, while `Peter,.N+firstName` only recognizes `Peter` and `PETER`. Lower and upper case letters are defined in the alphabet file passed to the `Dico`. as a parameter.

Respecting white space is a very simple rule: For each sequence in the text to be recognized by a dictionary entry, it has to have exactly the same number of spaces. For example, if the dictionary contains `aujourd'hui,.ADV`, the sequence `Aujourd' hui` will not be recognized because of the space that follows the apostrophe.

3.7 Bibliography

The table XXX gives some references for electronic dictionaries with simple and compound words. For more details, see the references page on the Unitex website[1]

XXXcaption Some bibliographical references for electronic dictionaries

[1] http://www-igm.univ-mlv.fr/~unitex

Chapter 4

Searching with regular expressions

This chapter describes how to search for simple patterns in a text by using regular expressions.

4.1 Definition

The goal of this chapter is not to give an introduction on formal languages but to show how to use regular expressions in Unitex in order to search for simple patterns. Readers who are interested in a more formal presentation can consult the many works that discuss regular expression patterns.

A regular expression can be:

- a lexical unit (`livre`) or a pattern (`<smoke.V>`);
- the concatenation of two regular expressions (`he smokes`);
- the union of two regular expressions (`Pierre+Paul`);
- the Kleene star of a regular expression (`finish*`).

4.2 Lexical units

In a regular expression a lexical unit is a sequence of letters. The symbols period , plus, star, less than as well as the opening and closing parentheses have a special meaning. It is therefore necessary to precede them with an escape character \ if you want to search for them. Here are some examples of valid lexical units:

```
cat
3\.1415
\(1984\)
{S}
```

By default Unitex is set up to let lower case patterns also find upper-case matches. It is possibe to enforce case-sensitive matching using quotation marks. Thus, `''peter''` recognizes only the form `peter` and not `Peter` or `PETER`.

NOTE: in order to make a space obligatory, you needs to be enclosed in quotation marks.

4.3 Patterns

4.3.1 Special symbols

There are two kinds of patterns. The first category contains all symbols that have been introduced in section 2.5.2 except for the symbol `<^>`, which matches a line feed. Since all line feeds have been replaced by spaces this symbol cannot longer be useful when searching for patterns. These symbols, also called *meta-symbols*, are the following:

- <E> : the empty word or epsilon. Matches the empty string;
- <TOKEN> : matches any lexical unit
- <MOT> : matches any lexical unit that consists of letters;
- <MIN> : matches any lower-case lexical unit;
- <MAJ> : matches any upper-case lexical unit;
- <PRE> : matches any lexical unit that consists of letters and starts with a capital letter.
- <DIC> : matches any word that is present in the dictionaries of the text;
- <SDIC> : matches any simple word in the text dictionaries;
- <CDIC> : matches any composed word in the dictionaries of the text;
- <NB> : matches any contiguous sequence of digit (1234 is matched but not 1 234);
- # : prohibits the presence of space.

4.3.2 References to the dictionaries

The second kind of patterns refers to the information in the dictionaries of the text.

The four possible forms are:

- `<be>`: matches all the entries that have `be` as canonical form;
- `<be.V>`: matches all entries having `be` as canonical form and the grammatical code `V`;
- `<V>`: matches all entries having the grammatical code `V`;

- `{am,be.V} or <am,be.V>`: matches all the entries having `am` as inflected form, `be` as canonical form and the grammatical code `V`. This kind of pattern is only of interest if applied to the text automaton where all the ambiguities of the words are explicit.

  While executing a search on the text that pattern matches the same as the simple lexical unit `am`.

4.3.3 Grammatical and semantic constraints

The reference to the dictionary (`V`) in these examples is elementary. It is possible to express more complex patterns by using several grammatical or semantic codes separated by the character `+`. An entry of the dictionary is then only found if it has all the codes that are present in the pattern. The pattern `<N+z1>` thus recognizes the entries:

```
broderies,broderie.N+z1:fp
capitales européennes,capitale européenne.N+NA+Conc+HumColl+z1:fp
```

but not:

```
Descartes,René Descartes.N+Hum+NPropre:ms
habitué,.A+z1:ms
```

It is possible to exclude codes by preceding them with the character `-` instead of `+`. In order to be recognized an entry has to contain all the codes authorized by the pattern and none of the prohibited codes. The pattern `<A-z3>` thus recognizes all the adjectives that do not have the code `z3` (cf. table 3.2). If you want to refer to a code containing the character `-` you have to escape this character by preceding it with a `\`. Thus, the pattern `<N+faux\-ami>` could recognize all entries of the dictionaries containing the codes `N` and `faux-ami`.

The order in which the codes appear in the pattern is not important. The three following patterns are equivalent:

```
<N-Hum+z1>
<z1+N-Hum>
<-Hum+z1+N>
```

NOTE: it is not possible to use a pattern that only has prohibited codes. `<-N>` and `<-A-z1>` are thus incorrect patterns.

4.3.4 Inflectional constraints

It is also possible to specify constraints about the inflectional codes. These constraints have to be preceded by at least one grammatical or semantic code. They are represented as inflectional codes present in the dictionaries. Here are some examples of patterns using inflectional constraints:

- `<A:m>` recognizes a masculine adjective;

- <A:mp:f> recognizes a masculine plural or a feminin adjective;
- <V:2:3> recognizes a verb in the 2nd or 3rd person; that excludes all tenses that have neither a 2nd or 3rd person (infinitive, past participle and present participle) as well as the tenses that are conjugated in the first person.

In order to let a dictionary entry E be recognized by pattern M, it is necessary that at least one inflectional code of E contains all the characters of an inflectional code of M. Consider the following example:

E=pretext,.V:W:P1s:P2s:P1p:P2p:P3p
M=<V:P3s:P3>

No inflectional code of E contains the characters P, 3 and s at the same time. However, the code P3p of E does contain the characters P and 3. The code P3 is included in at least one code of E, the pattern M thus recognizes the entry E. The order of the characters inside an inflectional code is without importance.

4.3.5 Negation of a pattern

It is possible to negate a pattern by placing the character ! immediately after the character <.

Negation is possible with the patterns <MOT>, <MIN>, <MAJ>, <PRE>, <DIC> as well as with the patterns that carry grammatical, semantic of inflectional codes (*i.e.* <!V-z3:P3>). The patterns # and " " are each the negation of the other. The pattern <!MOT> recognizes all lexical units that do not consist of letters except for the phrase separator.

The negation is interpreted in a special way in the patterns <!DIC>, <!MIN>, <!MAJ> and <!PRE>. Instead of recognizing all forms that are not recognized by the pattern without negation these patterns find only forms that are sequences of letters. Thus, the pattern <!DIC> allows to find all unknown words in a text. These unknown forms are mostly proper name, neologisms and spelling errors.

Here are some examples of patterns that mix the different types of constraints:

- <A-Hum:fs> : a non-human adjective in feminine singular;
- <lire.V:P:F> : the verb *lire* in present tense or future;
- <suis,suivre.V> : the word *suis* as inflected form of the verb *suivre* (as opposed the form of the verb *être*;
- <facteur.N-Hum> : all nominal entries that have *facteur* as canonical form and that do not have the semantic code Hum;
- <!ADV> : all words that are not adverbs;
- <!MOT> : all symbols that are not letters except for the phrase separator (cf. figure 4.2).

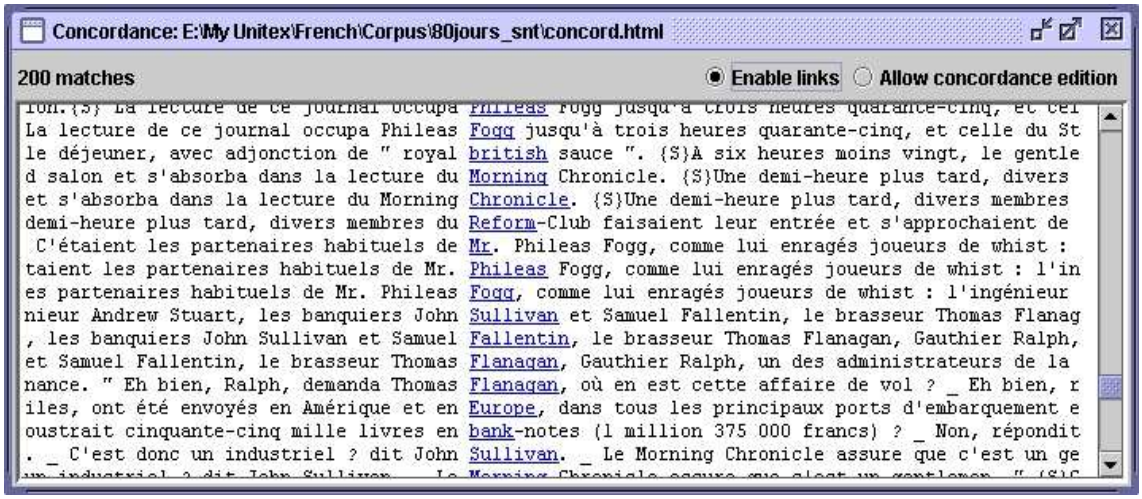

Figure 4.1: Result of the search for `<!DIC>`

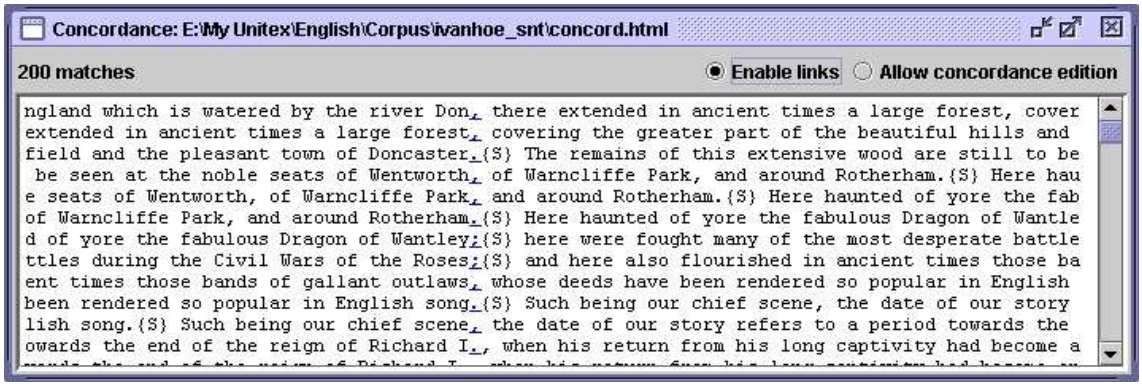

Figure 4.2: Result of a search for the pattern `<!MOT>`

4.4 Concatenation

There are three ways to concatenate regular expressions. The first consists in using the concatenation operator which is represented by the period. Thus, the expression:

```
<DET>.<N>
```

recognizes a determiner followed by a noun. The space can also be used for concatenation. The following expression:

```
the <A> cat
```

recognizes the lexical unit *the*, followed by an adjective and the lexical unit *cat*. Finally, it is possible to omit the period and the space before an opening bracket or the character < as

well as after a closing bracket or after the character >. The brackets are used as delimiters of a regular expression. All of the following expression are equivalent:

```
the <A> cat
(the <A>)cat
the.<A>.cat
(the)<A>.cat
(the(<A>))(cat)
```

4.5 Union

The union of regular expressions is expressed by putting the character + between them. The expression:

```
(I+you+he+she+it+we+they) <V>
```

recognizes a pronoun followed by a verb. If an element in an expression is optional it is sufficient to use the union of this element and the empty word epsilon.

Examples:

`the(little+<E>)cat` recognizes the sequences *the cat* and *the little cat*
`(<E>+Anglo-)(French+Indian)` recognizes *French*, *Indian*, *Anglo-French* and *Anglo-Indian*

4.6 Kleene star

The Kleene star, represented by the character *, allows to recognize zero, one or several occurrences of an expression. The star must be placed on the right hand side of the element in question. The expression:

```
this is very* cold
```

recognizes *this is cold*, *this is very cold*, *this is very very cold*, etc. The star has a higher priority than the other operators. You have to use brackets in order to apply the star to a complex expression. The expression:

```
0,(0+1+2+3+4+5+6+7+8+9)*
```

recognizes a zero followed by a comma and by a possibly empty sequence of digits.

ATTENTION: It is prohibited to search for the empty word with a regular expression. If you try to search for `(0+1+2+3+4+5+6+7+8+9)*`, the program will flag an error as shown in figure 4.3.

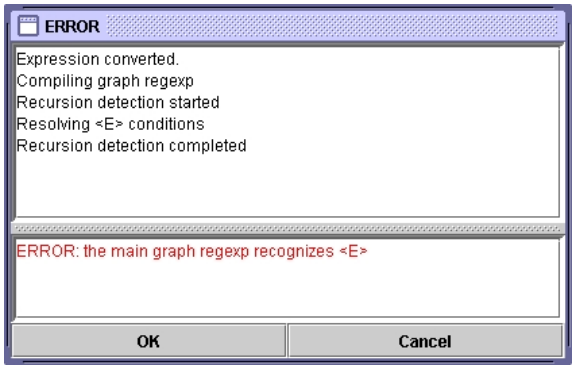

Figure 4.3: Error message when searching for the empty word

4.7 Morphological Filters

It is possible to apply morphological filters to the lexemes found. For that, it is necessary to immediately follow the lexeme found by a pattern in double angle brackets:

motif lexical<<*morphological pattern*>>

The morphological patterns are expressed as regular expressions in POSIX format (see [**?**] for the detailed syntax). Here are some examples of elementary filters:

- `<<ss>>`: contains `ss`
- `<<^a>>`: begins with `a`
- `<<ez$>>`: ends with `ez`
- `<<a.s>>`: contains `a` followed by any character, followed by `s`
- `<<a.*s>>`: contains `a` followed by a sequence of any character, followed by `s`
- `<<ss|tt>>`: contains `ss` or `tt`
- `<<[aeiouy]>>`: contains a non accentuated vowel
- `<<[aeiouy]{3,5}>>`: contains a sequence of non-accentuated vowels whose length is between 3 and 5
- `<<ée?>>`: contains `é` followed by an optional `e`
- `<<ss[^e]?>>`: contains `ss` followed by an optional character which is not `e`

It is possible to combine these elementary filters to form more complex filters:

- `<<[ai]ble$>>`: ends with `able` or `ible`

- `<<^(anti|pro)-?>>`: begins with `anti` or `pro`, followed by an optional dash
- `<<^([rst][aeiouy]){2,}$>>`: a word formed by 2 or more sequences beginning with `r`, `s` or `t` followed by a non-accentuated vowel
- `<<^([^l]|l[^e])>>`: doesn't begin with `l` unless the second letter is an `e`, in other words any word except the ones starting with `le`

By default, a morphological filter alone is regarded as applying it to the pattern `<TOKEN>`, that means any lexical pattern. On the other hand, when a filter follows a lexical pattern immediately, it applies to what was recognized by the lexical pattern. Here are some examples of such combinations:

- `<V:K><<i$>>`: Past participle ending with `i`
- `<CDIC><<->>`: A compound word containing a dash
- `<CDIC><< .* >>`: a compound word containing two spaces
- `<A:fs><<^pro>>`: a feminin singular adjective beginning with `pro`
- `<DET><<^([^u]|(u[^n])|(...+))>>`: a (French) determiner different from `un`
- `<!DIC><<es$>>`: a word which is not in the dictionary and which ends with `es`
- `<V:S:T><<uiss>>`: a verb in the past or present subjunctive, and containing `uiss`

NOTE: By default, morphological filters are subject to the same variations of case as the lexical patterns. Thus, the pattern `<<^é>>` will recognize all the words starting with `é`, but also those which start with `e` or `E`. To force the matcher to respect the exact case of the pattern, it is necessary to add `_f_` immediately after it. Example: `<A><<^é>>_f_`

4.8 Search

4.8.1 Configuration of the search

In order to search for an expression first open a text (cf. chapter 2). Then click on "Locate Pattern..." in the menu "Text". The window of figure 4.4 appears.

The box "Locate pattern in the form of" allows to select regular expression or grammar. Click on "Regular expression".

The box "Index" allows to select the recognition mode:

- "Shortest matches" : prefer short matches;
- "Longest matches" : prefer longer matches. This is the default;

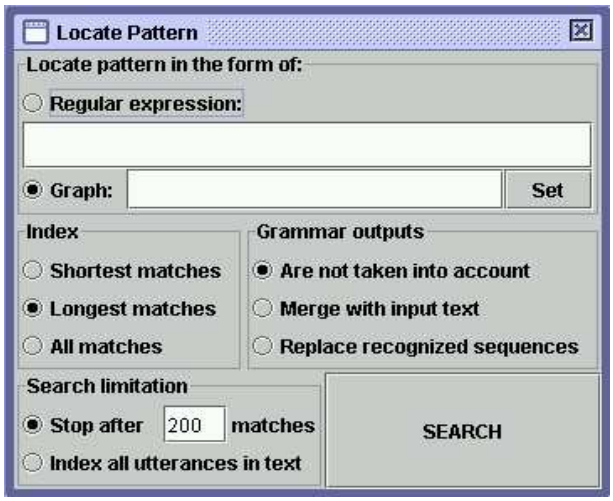

Figure 4.4: Window search for expressions

- "All matches" : Output all recognized sequences.

The box "Search limitation" is used to limit the number of results to a certain number of occurrences. By default, the search is limited to the 200 first occurrences.

The options of the box "Grammar outputs" do not concern regular expressions. They are described in section 6.6.

Enter an expression and click on "Search" in order to start the search. Unitex will transform the expression in a grammar in the format `.grf` . This grammar will then be compiled into a grammar of the format `.fst2` that will be used for the search.

4.8.2 Presentation of the results

When the search is finished, the window of figure 4.5 appears showing the number of matched occurrences, the number of recognized lexical entities and the ratio between this number and the total number of lexical units in the text.

After having clicked on "OK" you will see window 4.6 appear, which allows to configure the presentation of the matched occurrences. You can also open this window by clicking on "Display Located Sequences..." in the menu "Text". The list of occurrences is called a *concordance*.

The box "Modify text" offers the possibility to replace the matched occurrences with the generated outputs. This possibility will be examined in chapter 6.

Figure 4.5: Search results

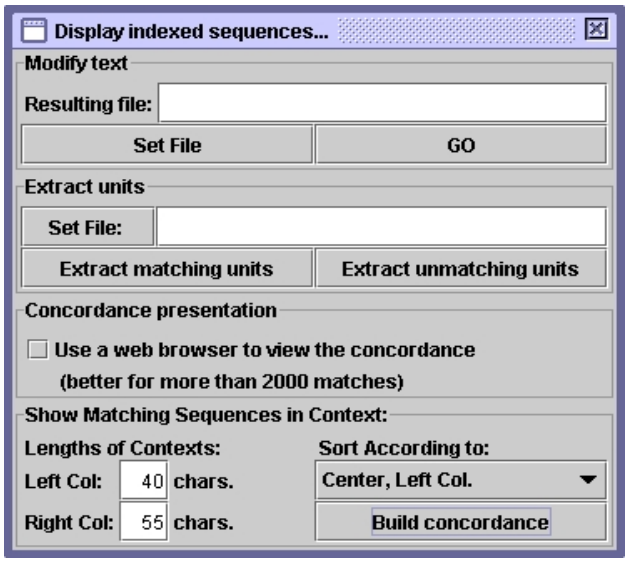

Figure 4.6: Configuration of the presentation of the found occurrences

The box "Extract units" allows to create a text file with all the sentences that do or do not contain matched units. The button "Set File" you can select the output file. Then click on "Extract matching units" or "Extract unmatching units" depending on whether you are interested in sentence with or without matching units.

In the box "Show Matching Sequences in Context" you can select the length in characters of the left and right contexts of the occurences that will be presented in the concordance. If an occurrence has less characters than its right context the line will be completed with the necessary number of characters. If an occurence has a length greater than that of the right context it will be displayed completely.

NOTE: in Thaï, the size of the contexts is measured in displayable characters and not in real characters. This makes it possible to keep the line alignment in the concordance despite the presence of diacritics that combine with other letters instead of being displayed as normal characters.

You can choose the sort order in the list "Sort According to". The mode "Text Order" displays the occurrences in the order of their appearance in the text. The six other modes allow sorting in columns. The three zones of a line are the left context, the occurrence and the right context. The occurrences and the right contexts are sorted from left to right. The left contexts are sorted from right to left. The default mode is "Center, Left Col.". The concordance is generated in the form of an HTML file.

If a concordance reaches several thousands of occurrences, it is advisable to display it in a web browser (Internet Explorer, Mozilla, Netscape, etc.) instead. Check the box "Use a web browser to view the concordance" (cf. figure 4.6). This option is activated by default if the number of occurrences is greater than 3000. You can configure which web browser to use by clicking on "Preferences..." in the menu "Info". Click on the tab "Text Presentation" and select the program to use in the field "Html Viewer" (cf. figure 4.7).

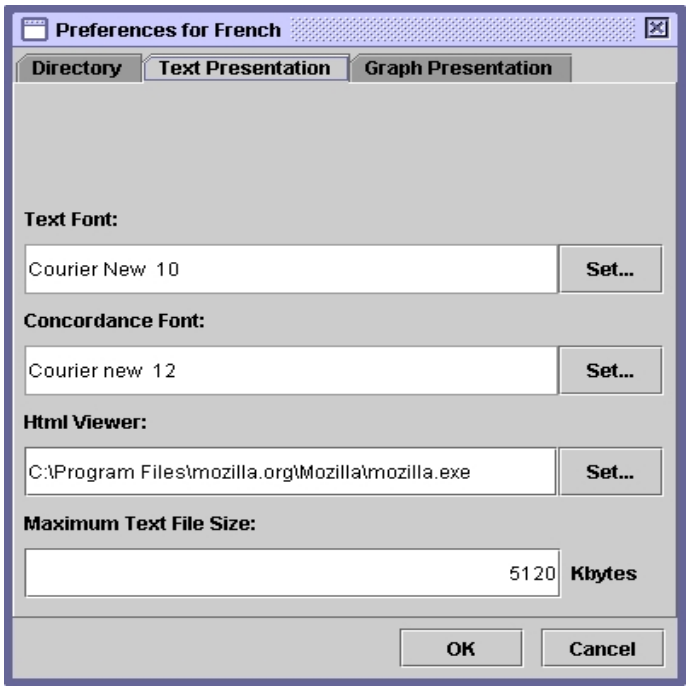

Figure 4.7: Selection of a web browser for displaying concordances

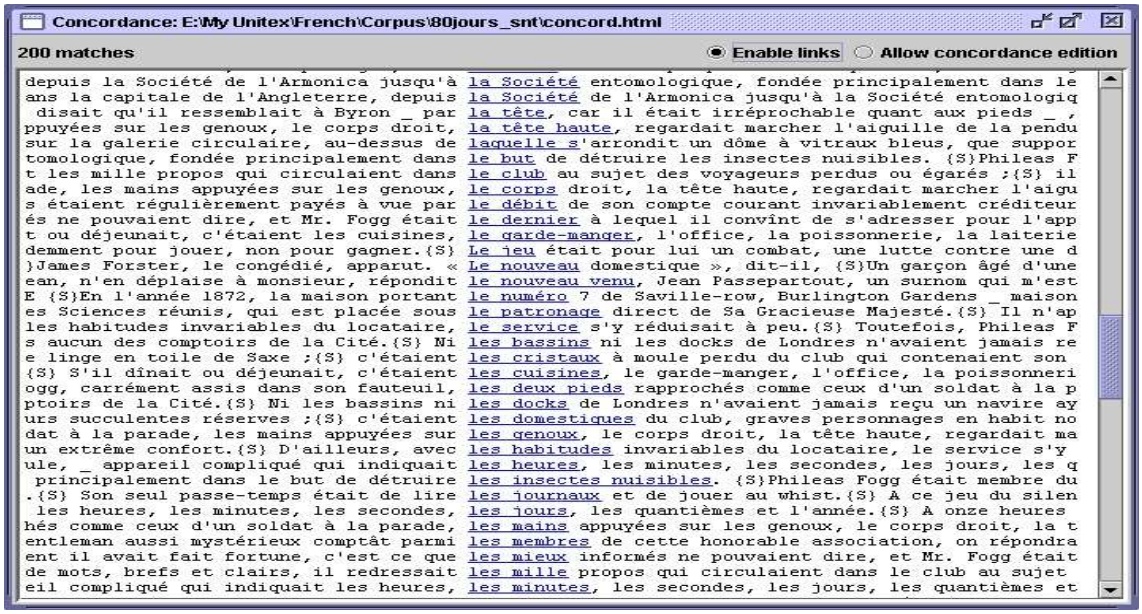

Figure 4.8: Example concordance

Chapter 5

Local grammars

Local grammars are a powerful tool to represent the majority of linguistic phenomena. The first section presents the formalism in which these grammars are represented. Then we will see how to construct and present grammars using Unitex.

5.1 The Local grammar formalism

5.1.1 Algebraic grammars

Unitex grammars are variants of algebraic grammars, also known as context-free grammars. An algebraic grammar consists of rewriting rules. Below you see a grammar that matches any number of a characters:

$S \rightarrow aS$
$S \rightarrow \varepsilon$

The symbols to the left of the rules are called *non-terminal symbols* since they can be replaced. Symbols that cannot be replaced by other rules are called *terminal symbols*. The items at the right side are sequences of non-terminal and terminal symbols. The epsilon symbol ε designates the empty word. In the grammar above, S is a non-terminal symbol and a a terminal (symbol). S can be rewritten as either an a followed by a S or as the empty word. The operation of rewriting by applying a rule is called *derivation*. We say that a grammar recognizes a word if there exists a sequence of derivations that produces that word. The non-terminal that is the starting point of the first derivation is called an *axiom*.

The grammar above also recognizes the word *aa*, since we can derive this word according to the axiom S by applying the following derivations:

Derivation 1: rewriting the axiom to aS
$\underline{S} \rightarrow aS$

Derivation 2: rewriting S at the right side of aS
$S \rightarrow a\underline{S} \rightarrow aaS$

Derivation 3: rewriting S to ε
$S \rightarrow aS \rightarrow aa\underline{S} \rightarrow aa$

We call the set of words recognized by a grammar the *grammar of the language*. The languages recognized by algebraic grammars are called *algebraic languages*

5.1.2 Extended algebraic grammars

Extended algebraic grammars are algebraic grammars where the members on the right side of the rule are not just sequences of symbols but rational expressions.. Thus, the grammar that recognizes a sequence of an arbitrary number of a's can be written as a grammar consisting of one rule:

$S \rightarrow a^*$

These grammars, also called *recursive transition networks* (*RTN*) or *syntax diagrams*, are suited for a user-friendly graphical representation. Indeed, the right member of a rule can be represented as a graph whose name is the left member of the rule.

However, Unitex grammars are not exactly extended algebraic grammars, since they contain the notion of *transduction*. This notion, which is derived from the field of finite state automata, enables a grammar to produce some output. With an eye towards clarity, we will use the terms grammar or graph. When a grammar produces outputs, we will use the term *transducer*, as an extension of the definition of a transducer in the area of finite state automata.

5.2 Editing graphs

5.2.1 Import of Intex graphs

In order to be able to use Intex graphs in Unitex, they have to be converted to Unicode. The conversion procedure is the same as the one for texts (see section 2.2). If you're using Microsoft Word to perform this conversion, make sure that the graph always has the `.grf` extension after the conversion, since it happens that the `.txt` extension is automatically appended. If a `.txt` extension was appended, it has to be removed.

ATTENTION: A graph converted to Unicode that was used in Unitex cannot be used in Intex any longer.

In order to use it again in Intex, you have to convert the text to ASCII, for example using the `Uni2Asc` program. In addition to this, you have to open the graph in a text editor and replace the first line:

```
#Unigraph
```

by the following line:

```
#FSGraph 4.0
```

5.2.2 Creating a graph

In order to create a graph, click on "New" in the "FSGraph" menu. You will then see the window coming up as in figure 5.2. The symbol in arrow form is the *init state* of the graph. The round symbol with a square is the *final state* of the graph. The grammar only recognizes expressions that are described along the paths between init and final state.

Figure 5.1: FSGraph menu

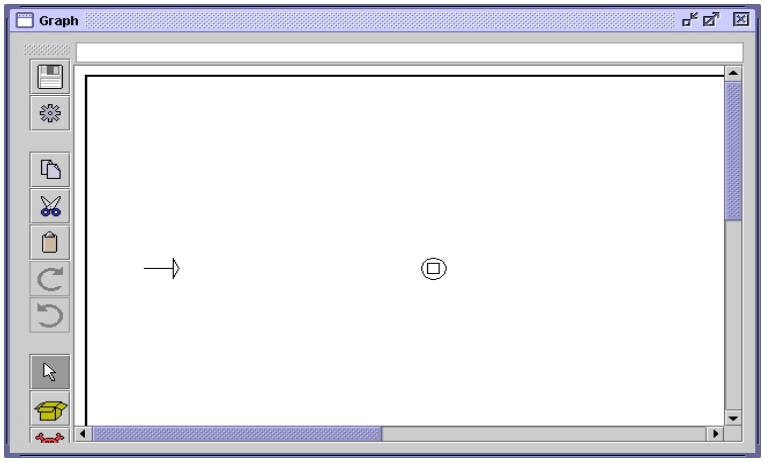

Figure 5.2: Empty graph

In order to create a box, click inside the window while pressing the Ctrl key. A blue rectangle will appear that symbolizes the empty box that was created (see figure 5.3). After creating the box, it is automatically selected.

You see the contents of that box in the text field at the top of the window. The newly created box contains the <E> symbol that represents the empty word epsilon. Replace this

symbol by the text `le+la+l'+les` and press the enter key. You see that the box now contains four lines (see figure 5.4). The + character serves as a separator.. The box is displayed in the form of red text lines since it is not connected to another one at the moment. We often use this type of boxes to insert comments into a graph.

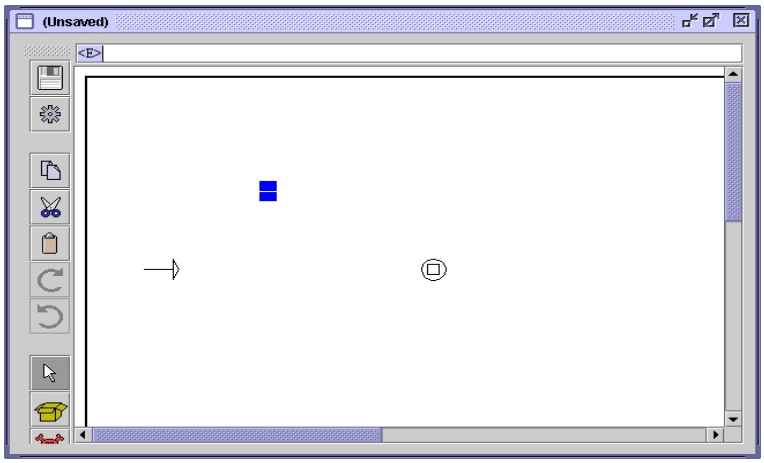

Figure 5.3: Creating a box

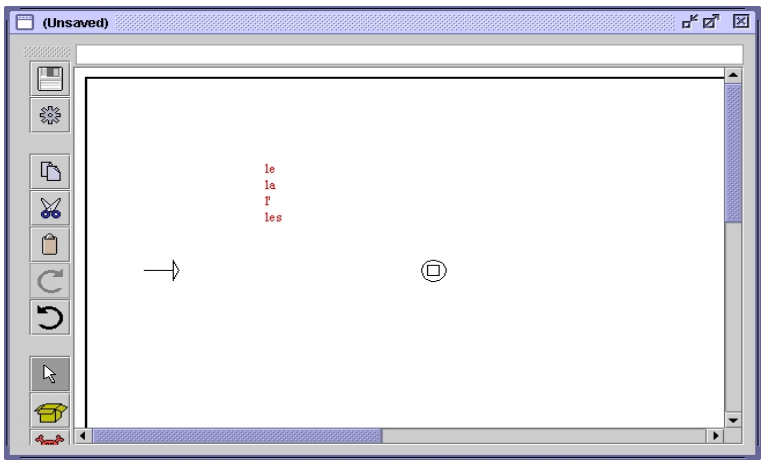

Figure 5.4: Box containing `le+la+l'+les`

To connect a box to another one, first click on the source box, followed by a click on the target box.

If there already exists a transition between two boxes, it is deleted. It is also possible to use this operation by clicking first on the target box and then on the source box while pressing Shift. In our example, after connecting the box to the init and the final states of the graph, we get a graph as in figure 5.5:

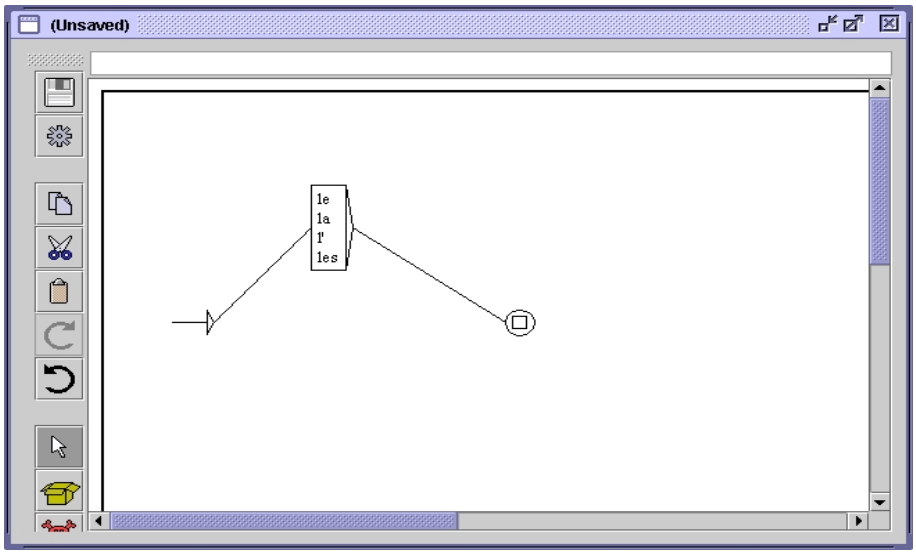

Figure 5.5: Graph that recognizes pronouns in French

NOTE: If you double-click a box, you connect this box to itself (see figure 5.6). To undo this double-click on the same box a second time.

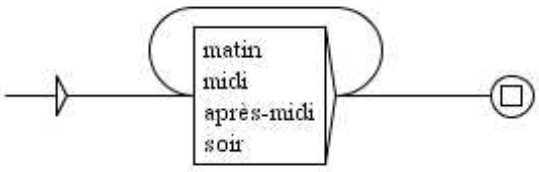

Figure 5.6: Box connected to itself

Click on "Save as..." in the "FSGraph" menu to save the graph.

By default, Unitex proposes to save the graph in the sub-directory `Graphs` in your personal folder. You can see if the graph was modified after the last saving if the title contains the text `(Unsaved)`.

5.2.3 Sub-Graphs

In order to call a sub-graph, its name is inserted into a box and preceded by the `:` character. If you enter the text `alpha+:beta+gamma+:e:\Greek\delta.grf` into a box, you get a box similar to the one in figure 5.7:

You can indicate the complete name of the graph (`e:\grec\delta.grf`) or simply the name in the access path (`beta`); in this case, the the sub-graph is expected to be in the same directory as the graph that references it.

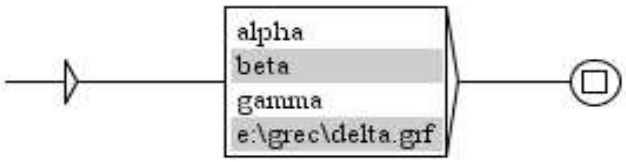

Figure 5.7: Graph that calls sub-graphs `beta` and `delta`

Calls to these sub-graphs are represented in the boxes by gray lines . On Windows you can open a sub-graph by clicking on the gray line while pressing the Alt key. On Linux, the combination <Alt+Click> is intercepted by the system. In order to open a sub-graph, click on its name by pressing the left and the right mouse button simultaneously.

5.2.4 Manipulating boxes

You can select several boxes using the mouse. In order to do so, click and drag the mouse without releasing the button. When you release the button, all boxes touched by the selection rectangle will be selected and are displayed in white on blue ground:

Figure 5.8: Selecting multiple boxes

When the boxes are selected, you can move them by clicking and dragging the cursor without releasing the button. In order to cancel the selection, click on an empty area of the graph. If you click on a box, all boxes of the selection will be connected to it.

You can perform a copy-paste using several boxes. Select them and press <Ctrl+C> or click on "Copy" in the "Edit" menu. The selection is now in the Unitex clipboard.
You can then paste this selection by pressing <Ctrl+V> or by selecting "Paste" in the "Edit" menu.

NOTE: You can paste a multiple selection into a different graph than the one where you copied it from.

In order to delete boxes, select them and delete the text that they contain. Delete the text presented in the text field above the window and press the enter key. The init and final states cannot be deleted.

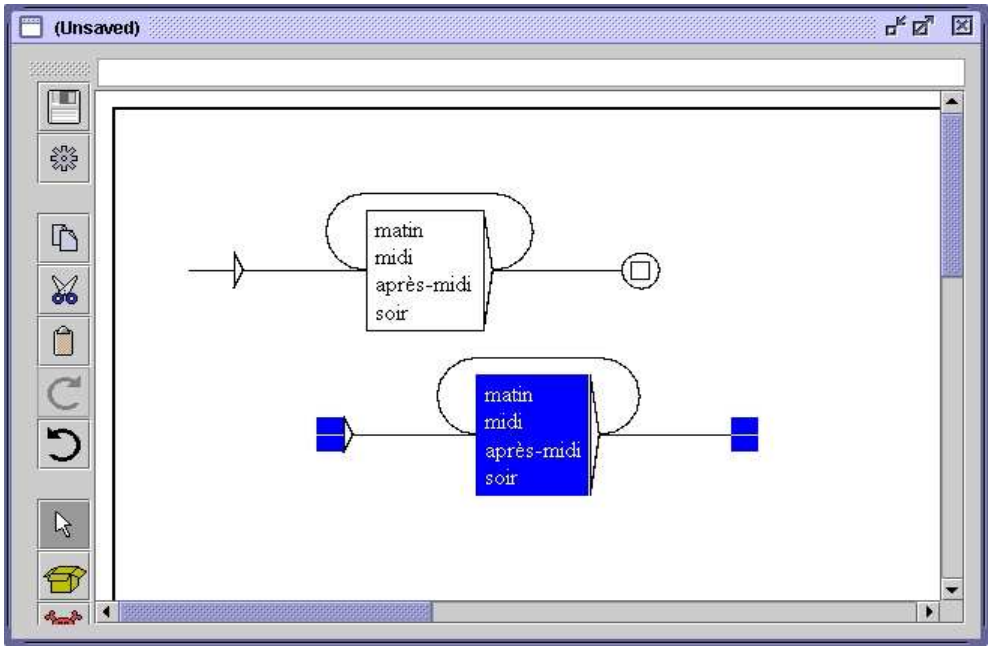

Figure 5.9: Copy-Paste of a multiple selection

5.2.5 Transducers

A transduction is an output associated with a box. To insert a transduction, use the special character `/`. All characters to the right of it will be part of the transduction. Thus, the text `un+deux+trois/nombre` results in a box like in figure 5.10:

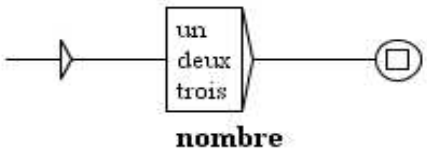

Figure 5.10: Example of a transduction

The transduction associated with a box is represented in bold text below it.

5.2.6 Using Variables

It is possible to select parts of a recognized text by a grammar using variables. To associate a variable `var1` with parts of a grammar, use the special symbols `$var1(` and `$var1)` to define the beginning and the end of the part to store. Create two boxes containing one `$var1(` and the second `$var1)`. These boxes must not contain anything but the variable

name preceded by $ and followed by a parenthesis. Then link these boxes to the zone of the grammar to store. In the graph in figure 5.11 you see a sequence beginning with an upper case letter after `Mister` or `Mr.`. This sequence will be stored in a variable named `var1`.

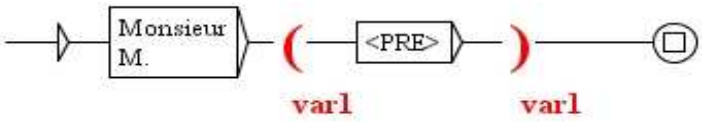

Figure 5.11: Using the variable `var1`

The variable names may contain letters (without accents), upper or lower case, numbers, or the _ (underscore) character. Unitex distinguishes between uppercase and lowercase characters.

When a variable is defined, you can use it in transductions by preceding its name with $. The grammar in figure 5.12 recognizes a date formed by a month and a year, and produces the same date as an output, but in the order year-month.

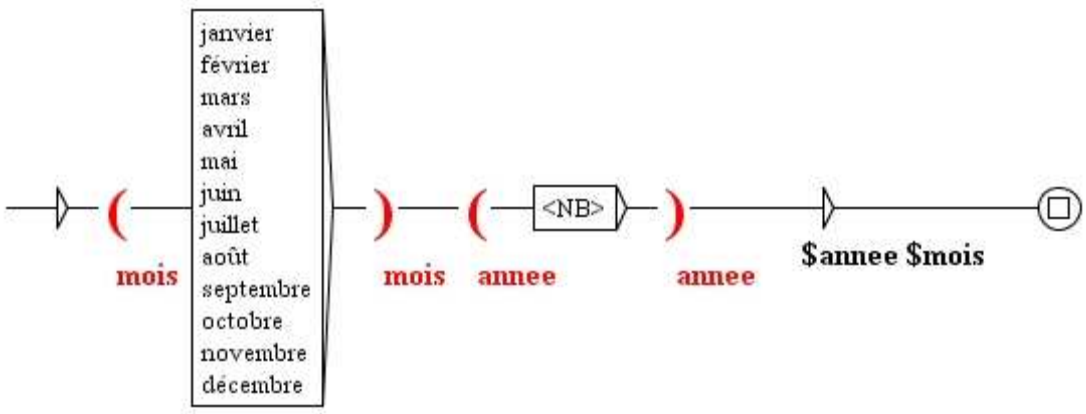

Figure 5.12: Inverting month and year in a date

5.2.7 Copying Lists

It can be practical to perform a copy-paste operation on a list of words or expressions from a text editor to a box in a graph. In order to avoid having to copy every term manually, Unitex provides a means to copy lists. To use this, select the list in your text editor and copy it using <Ctrl+C> or the copy function integrated in your editor. Then create a box in your graph, and press <Ctrl+V> or use the "Paste" command in the "Edit" menu to paste it into the box. A window as in figure 5.13 opens:

Figure 5.13: Selecting a context for copying a list

This window allows you to define the left and right contexts that will automatically be used for each term of the list. By default, these contexts are empty. If you use the contexts `<` and `.V>` with the following list:

eat
sleep
drink
play
read

you will get the box in figure 5.14:

Figure 5.14: Box resulting from copying a list and applying contexts

5.2.8 Special Symbols

The Unitex graph editor interprets the following symbol in a special manner:

```
" + : / < > # \
```

Table 5.1 summarizes the meaning of these symbols for Unitex, as well as the places where these characters are recognized in the texts

Caracter	Meaning	Escape
"	quotation marks mark sequences that must not be interpreted by Unitex, and whose case must be taken verbatim	\"
+	+ separates different lines within the boxes	"+"
:	: introduces a call to a subgraph	":" ou \:
/	/ indicates the start of a transduction within a box	\/
<	< indicates the start of a pattern or a meta	"<" ou \<
>	> indicates the end of a pattern or a meta	">" ou \>
#	# prohibits the presence of a space	"#"
\	\ escapes most of the special characters	\\

Table 5.1: encoding of special characters in the graph editor

5.2.9 Toolbar Commands

The toolbar to the left of the graphs contains short cuts for certain commands and allows to manipulate boxes of a graph by using some "utilities". This toolbar may be moved by clicking on the "rough" zone. It may also be dissociated from the graph and appear in an separate window (see figure 5.15). In this case, closing this window puts the toolbar back at its initial position. Each graph has its own toolbar.

Figure 5.15: Toolbar

The first two icons are shortcuts for saving and compiling the graph. The five following correspond to the Copy, Cut, Paste, Redo and Undo operations. The last icon showing a key is a shortcut to open the window with the graph display options.

The other 6 icons correspond to edit commands for boxes. The first one, a white arrow, corresponds to the boxes' normal edit mode. The 5 others correspond to specific utilities. In order to use a utility, click on the corresponding icon: The mouse cursor changes its form and mouse clicks are then interpreted in a particular fashion. What follows is a description of these utiltities, from left to right:

- creating boxes: creates a box at the empty place where the mouse was clicked;
- deleting boxes: deletes the box that you click on;

- connect boxes to another box: using this utility you select one or more boxes and connect it or them to another one. In contrast to the normal mode, the connections are inserted to the box where the mouse button was released on;

- connect boxes to another box in the opposite direction: this utility performs the same operation as the one described above, but connects the boxes to the one clicked on in opposite direction;

- open a sub-graph: opens a sub-graph when you click on a grey line within a box.

5.3 Display options

5.3.1 Sorting the lines of a box

You can sort the contents of a box by selecting it and clicking on "Sort Node Label" in the "Tools" submenu of the "FSGraph" menu. This sort operation doesn't use the `SortTxt` program. It uses a basic sort mechanism that sorts the lines of the box according to the order of the characters in the Unicode encoding.

5.3.2 Zoom

The "Zoom" submenu allows you to choose the zoom scale that is applied to display the graph.

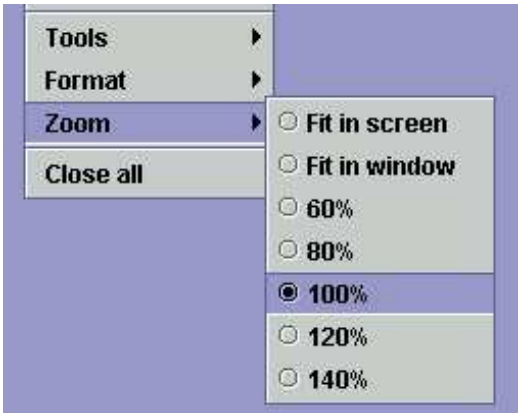

Figure 5.16: Zoom Sub-Menu

The option "Fit in screen" stretches or shrinks the graph in order to fit it into the screen. The option "Fit in window" adjusts the graph for it to be displayed completely in the window.

5.3.3 Antialiasing

Antialiasing is a shading effect that avoids pixelisation effects. You can activate this effect by clicking on "Antialiasing..." in the "Format" sub-menu. Figure 5.17 shows one graph displayed normally (the graph on top) and with antialiasing (the graph at the bottom).

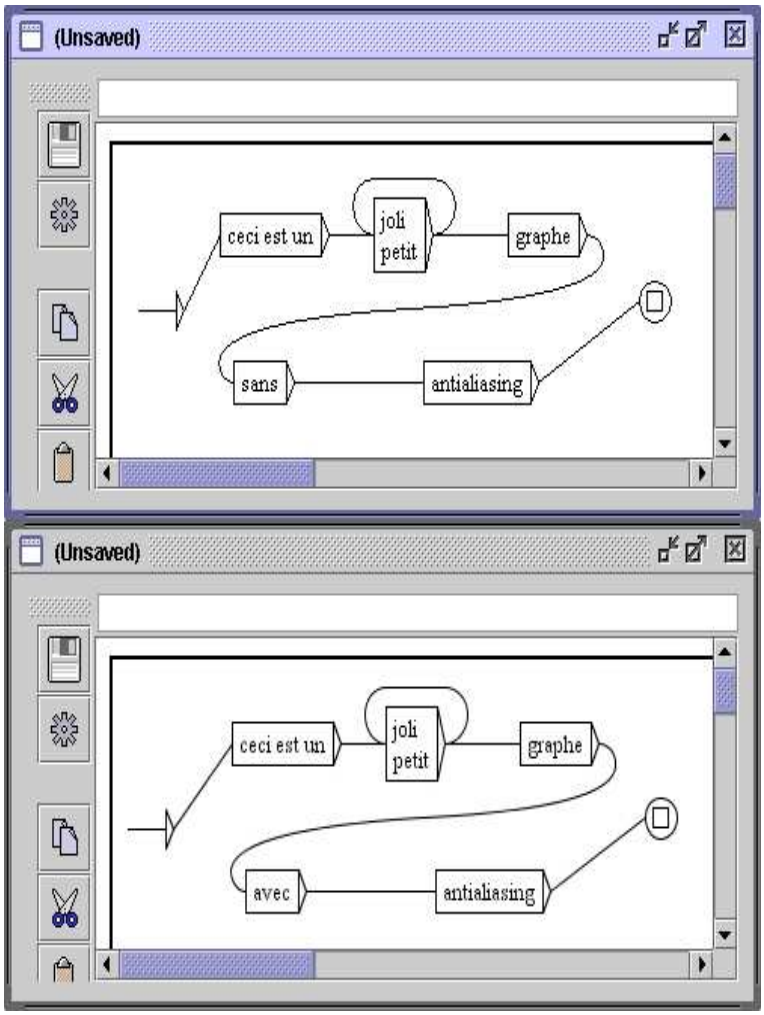

Figure 5.17: Antialiasing example

This effect slows Unitex down. We recommend not to use it if your machine is not powerful enough.

5.3.4 Box alignment

In order to get nice-looking graphs, it is useful to align the boxes, both horizontally and vertically. To do this, select the boxes to align and click on "Alignment..." in the "Format"

sub menu of the "FSGraph" menu or press <Ctrl+M>. You will then see the window in figure 5.18.

The possibilities for horizontal alignment are:

- Top: The boxes are aligned with the top-most box;
- Center: The boxes are centered with the same axis;
- Bottom: The boxes are aligned with the bottom-most box.

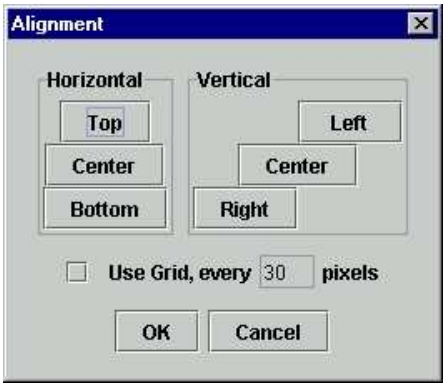

Figure 5.18: Alignment window

The possibilities for vertical alignment are:

- Left: The boxes are aligned with the left-most box;
- Center: The boxes are centered with the same axis;
- Right: The boxes are aligned with the right-most box.

Figure 5.19 shows an example of alignment. The group of boxes to the right is a copy of the ones to the left that was aligned vertically to the left.

The option "Use Grid" in the alignment window shows a grid as the background of the graph. This allows to approximately align the boxes.

5.3.5 Display, Options and Colors

You can configure the display style of a graph by pressing <Ctrl+R> or by clicking on "Presentation..." in the "Format" sub-menu of the "FSGraph" menu, which opens the window as in figure 5.21.

The font parameters are:

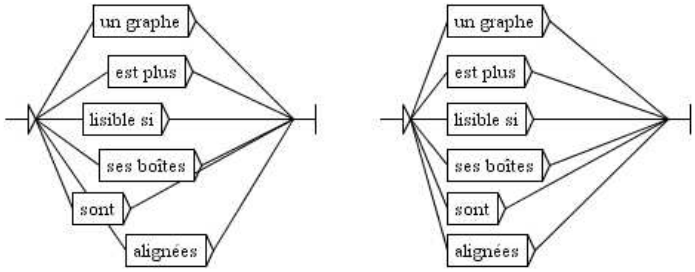

Figure 5.19: Example of aligning vertically to the left

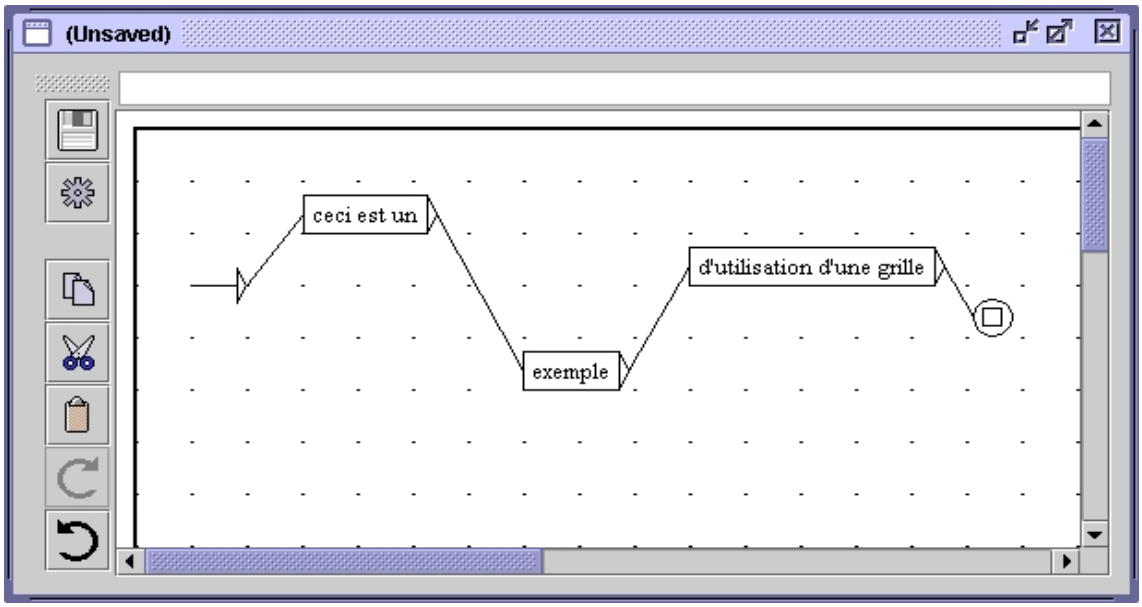

Figure 5.20: Example of using the grid

- Input: Font used within the boxes and in the text area where the contents of the boxes is edited;
- Output: font used for the attached transductions.

The color parameters are:

- Background: the background color;
- Foreground: the color used for the text and for the box display;
- Auxiliary Nodes: the color used for calls to sub-graphs;
- Selected Nodes: the color used for selected boxes;

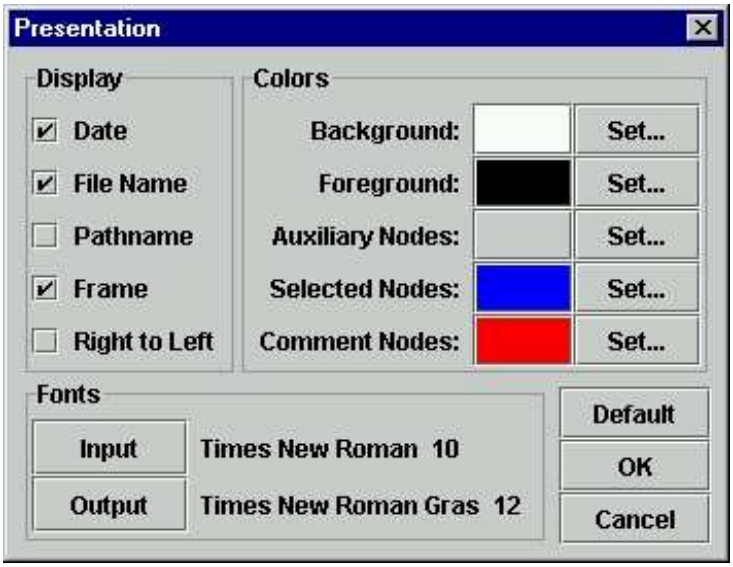

Figure 5.21: Configuring the display options of a graph

- Comment Nodes: the color used for boxes that are not connected to others.

The other parameters are:

- Date: display of the current date in the lower left corner of the graph;
- File Name: display of the graph name in the lower left corner of the graph;
- Pathname: display of the graph name along with its complete path in the lower left corner of the graph. This option only has an effect if the option "File Name" is selected;
- Frame: draw a frame around the graph;
- Right to Left: invert the reading direction of the graph (see an example in figure 5.22).

You can reset the parameters to the default ones by clicking on "Default". If you click on "OK", only the current graph will be modified. . In order to modify the preferences for a language as a default, click on "Preferences..." in the "Info" menu and choose the tab "Graph Representation".

The preferences configuration window has an extra option concerning antialiasing (see figure 5.23). This option activates antialiasing by default for all graphs in the current language. It is advised to not activate this option if your machine is not very fast.

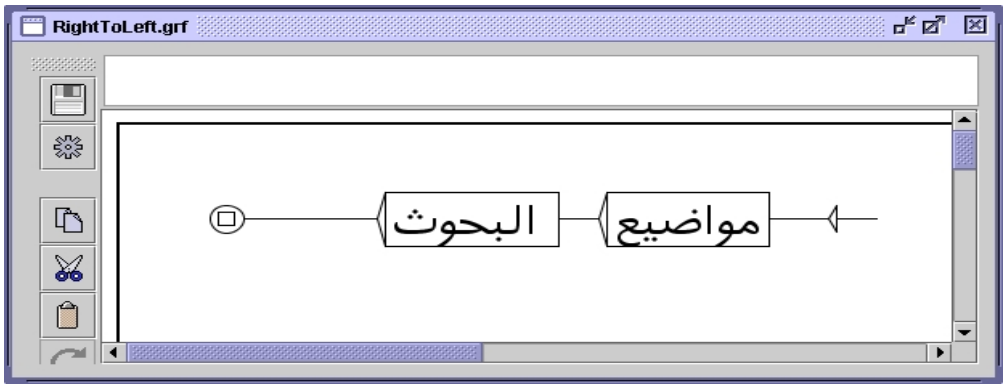

Figure 5.22: Graph with reading direction set to right to left

Figure 5.23: Default preferences configuration

5.4 Graphs outside of Unitex

5.4.1 Inserting a graph into a document

In order to include a graph into a document, you have to convert it to an image. To do this, activate antialiasing for the graph that interests you (this is not obligatory but results in a better image quality).

In Windows:

Press "Print Screen" on your keyboard. This key should be next to the F12 key. Start the `Paint` program in the Windows "Utilities" menu. Press <Ctrl+V>. `Paint` will tell you that the image in the clipboard is too large and asks if you want to enlargen the image. Click on "Yes". You can now edit the screen image. Select the area that interests you. To do so, switch to the select mode by clicking on the dashed rectangle symbol in the upper left corner of the window. You can now select the area of the image using the mouse. When you have selected the zone, press <Ctrl+C>. Your selection is now in the clipboard, you can now just go to your document and press <Ctrl+V> to paste your image.

In Linux:

Take a screen capture (for example using the program `xv`). Edit your image at once using a graphic editor (for example `TheGimp`), and paste your image in your document in the same way as in Windows.[1]

5.4.2 Printing a Graph

You can print a graph by clicking on "Print..." in the "FSGraph" menu or by pressing <Ctrl+P>.

ATTENTION: You should make sure that the page orientation parameter (portrait or landscape) corresponds to the orientation of your graph.

You can specify the printing preferences by clicking on "Page Setup" in the "FSGraph" menu. You can also print all open graphs by clicking on "Print All...".

[1] For those who want to get a vector graphic (small and scalable): (1) Use the Unitex Print Graph menu and print the graph to a Postscript file. (2) Clean the Postscript by typing `gs -sDEVICE=pswrite -dNOPAUSE -dBATCH -sOutputFile=clean.ps graph.ps` in your shell. Know you get a smaller file. Have a look on it using `gv`. (3) Now you can convert the graph with `convert` into various image formats.

Chapter 6

Advanced use of graphs

6.1 Types of graphs

Unitex can work with four types of graphs that correspond to the following uses: automatic inflection of dictionaries, preprocessing of texts, normalization of text automata and search for patterns. These different types of graphs are not interpreted in the same way by Unitex. Certain operations like the transduction are allowed for some types and forbidden for others. In addition, the special symbols are not the same depending on the type of the graph. This section presents each type of graph and shows their peculiarities.

6.1.1 Inflection graphs

An inflection graph describes the morphological variation that is associated with a word class by assigning inflectional codes to each variant. The paths of such a graph describe the modifications that have to be applied to the canonical forms so that the transduction contains the inflectional information that will be produced.

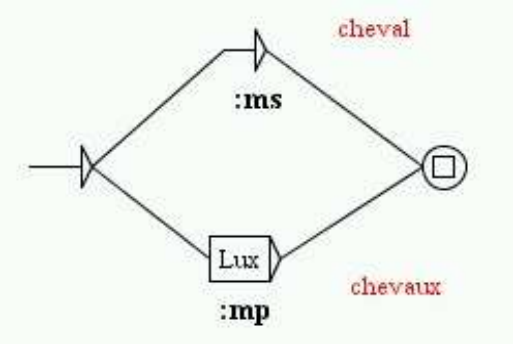

Figure 6.1: Example of an inflectional grammar

The paths may contain operators and letters. The possible operators are represented by the characters `L`, `R` and `C`. All letters that are not operators are characters. The only

allowed special symbol is the empty word <E>. It is not possible to refer to dictionaries in an inflection graph. It is also impossible to reference subgraphs.

Transductions are concatenated in order to produce a string of characters. This string is then appended to the line of the produced dictionary (cf. chapter 3.4). The transductions with variables do not make sense in an inflection graph.

The contents of an inflection graph are manipulated without a change of case: the lowercase letters stay lowercase, the same for the uppercase letters. Besides, the connection of two boxes is exactly equivalent to the concatenation of their contents together with the concatenation of their transductions. (cf. figure 6.2).

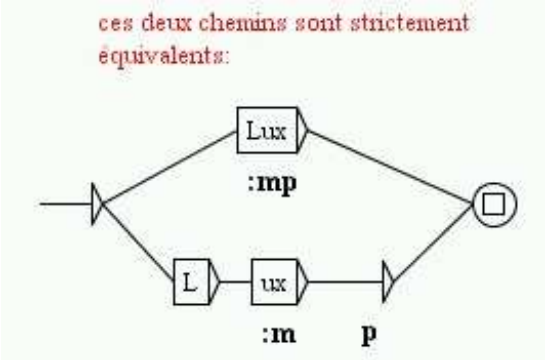

Figure 6.2: Two equivalent paths in an inflection grammar

The inflection graphs have to be compiled before being used by the inflection program.

6.1.2 Preprocessing graphs

Preprocessing graphs are meant to be applied to texts before they are tokenized into lexical units. These graphs can be used for inserting or replacing sequences in the texts. The two normal uses of these graphs are normalization of non-ambiguous forms and sentence boundary recognition.

The interpretation of these graphs in Unitex is very close to that of syntactic graphs used by the search for patterns. The differences are the following:

- you can use the special symbol <^> that recognizes a newline;
- it is impossible to refer to dictionaries;
- it is necessary to compile these graphs before they can be used for preprosessing operations.

The figures 2.9 and 2.10 show examples of preprocessing graphs.

6.1.3 Graphs for normalizing the text automaton

The graphs for normalization of the text automaton allow to normalize ambiguous forms. In fact they can describe several labels for the same form. These labels are then inserted into the text automaton thus making the ambiguities explicit. Figure 6.3 shows an extract of the normalization graph used for French.

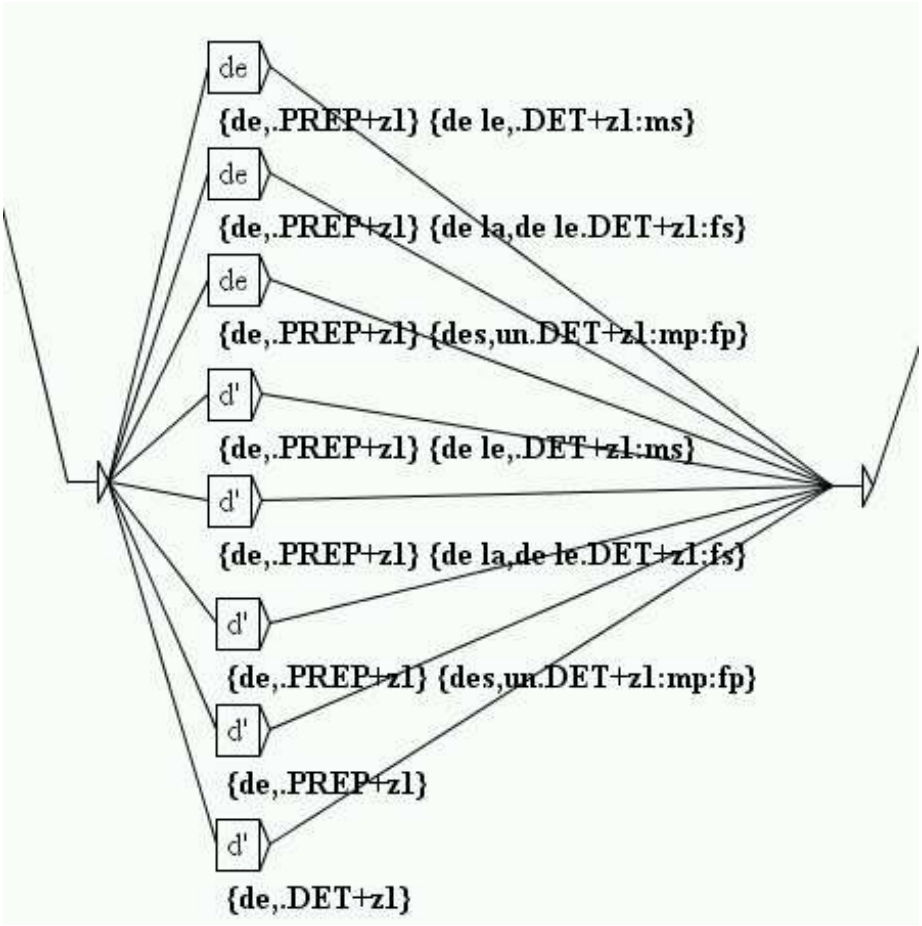

Figure 6.3: Extract of the normalisation graph used for French

The paths describe the forms that have to be normalized. Lower case and upper case variants are taken into account according to the following principle: uppercase letters in the graph only recognize uppercase letters in the text automaton; lowercase letters can recognize the lowercase and uppercase letters.

The transductions represent the sequence of the labels that will be inserted into the text automaton. These labels can be dictionary entries or strings of characters. The labels that represent entries of the dictionary have to respect the format for entries of a DELAF and are enclosed by the symbols { and }. The transductions with variables do not make sense in this kind of graph.

It is possible to reference subgraphs. It is not possible to reference dictionaries in order to describe the forms to normalize. The only special symbol that is recognized in this type of graph is the empty word `<E>`. The graphs for normalizing ambiguous forms need to be compiled before using them.

6.1.4 Syntactic graphs

The syntactic graphs, often called local grammars, allow to describe syntactic patterns that can then be searched in the texts. Of all kinds of graphs these have the greatest expressional power because they allow to refer to dictionaries.

Lower case/upper case variants may be used according to the principle described above. It is still possible to enforce respect of case by enclosing an expression in quotes. The use of quotes also allows to enforce the respect of spaces. In fact, Unitex by default assumes that a space is possible between two boxes. In order to enforce the presence of a space you have to enclose it in quotes. For prohibiting the presence of a space you have to use the special symbol #.

The syntactic graphs can reference subgraphs (cf. section 5.2.3). They also have transductions including transductions with variables. The produced sequences are interpreted as strings of characters that will be inserted in the concordances or in the text if you want to modify it. (cf. section 6.6.3).

The special symbols that are supported by the syntactic graphs are the same that are usable in the regular expressions. (cf. section 4.3.1).

It is not obligatory to compile the syntactic graphs before using them for pattern searching. If a graph is not compiled the system will compile it automatically.

6.1.5 ELAG Grammars

The syntax of grammars to resolve ambiguities est presented in section XXX.

6.1.6 Template graphs

The template graphs are meta-graphs that allow to generate a family of graphs starting from a lexical-grammar table. It is possible to construct model graphs for all possible kinds of graphs. The construction and use of model graphs will be explained in chapter 8.

6.2 Compilation of a grammar

6.2.1 Compilation of a graph

The compilation is the operation that converts the format `.grf` to a format that can be manipulated more easily by the Unitex programs. In order to compile a graph you open it and then click on "Compile FST2" in the submenu "Tools" of the menu "FSGraph". Unitex then starts the program `Grf2Fst2`. You can keep track of its execution in a window (cf. figure 6.4).

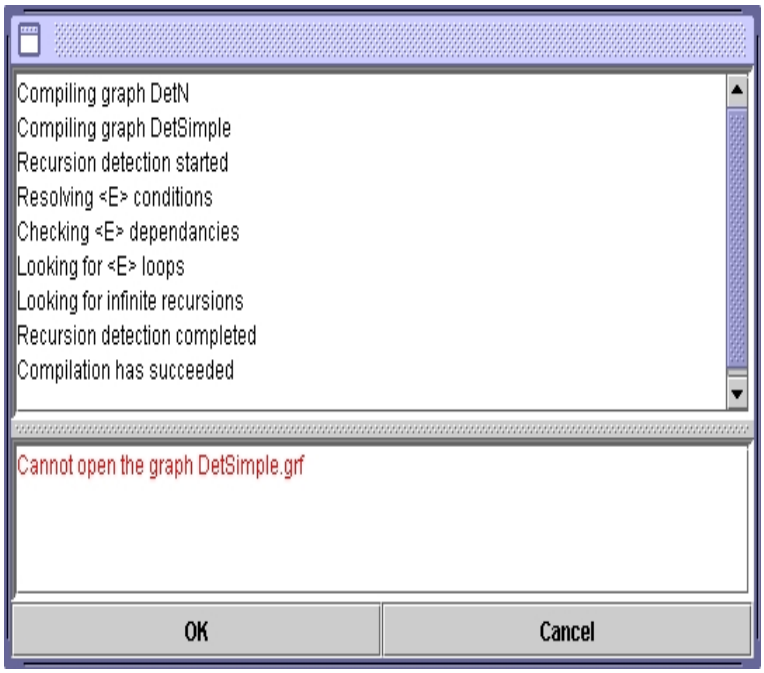

Figure 6.4: Compilation window

If the graph references subgraphs, those are automatically compiled. The result is a `.fst2` file that contains all the graphs that make up a grammar. The grammar is then ready to be used by the different Unitex programs.

6.2.2 Approximation with a finite state transducer

The FST2 format conserves the architecture in subgraphs of the grammars, which is what makes them different from strict finite state transducers. The program `Flatten` allows to transform a grammar FST2 in a finite state transducer whenever this is possible and to construct an approximation if not. This function thus permits to obtain objects that are easier to manipulate and to which all classical algorithms on automata can be applied.

In order to compile and thus transform a grammar select the command "Compile & Flatten FST2" in the submenu "Tools" of the menu "FSGraph". The window of figure 6.5 allows you to configure the operation of approximation.

The box "Flattening depth" lets you specify the level of embedding of subgraphs. This value represents the maximum depth up to which the calling of subgraphs will be replaced the subgraphs themselves.

The box "Expected result grammar format" allows to determine the behavior of the program beyond the selected limit. If you select the option "Finite State Transducer", the calls to

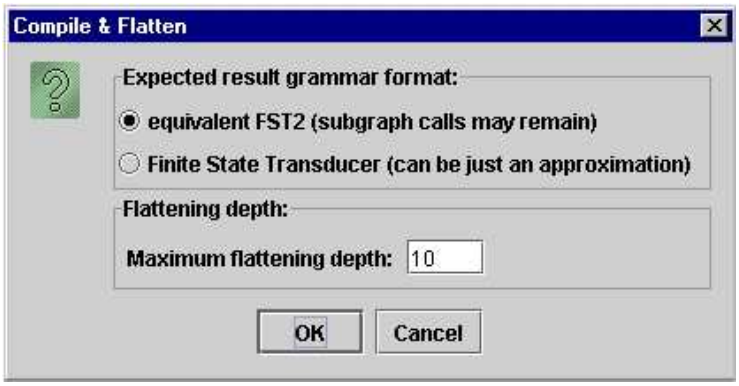

Figure 6.5: Configuration of approximation of a grammar

subgraphs will be ignored beyond the maximum depth. This option guarantees that we obtain a finite state transducer, however possibly not equivalent to the original grammar. On the contrary, the option "equivalent FST2" indicates that the program should allow for subgraph calls beyond the limited depth. This option guarantees the strict equivalence of the result with the original grammar but does not necessarily produce a finite state transducer. This option can be used for optimizing certain grammars.

A message indicates at the end of the approximation process if the result is a finite state transducer or an FST2 grammar and in the case of a transducer if it is equivalent to the original grammar. (cf. figure 6.6).

6.2.3 Constraints on grammars

With the exception of inflection grammars, a grammar can never have an empty path. This means that the principal path of a grammar must not recognize the empty word but this does not prevent a subgraph of that grammar from recognizing epsilon.

It is not possible to associate a transduction with a call to a subgraph. Such transductions are ignored by Unitex. It is therefore necessary to use an empty box that is situated to the left of the call to the subgraph in order to specify the transduction (cf. figure 6.7).

The grammars must not contain infinite loops because the Unitex programs cannot terminate the exploration of such a grammar. These infinite loops can originate from transitions that are labeled by the empty word or from recursive calls to subgraphs.

The infinite loops due to transitions with the empty word can have two origins of which the first is illustrated by the figure 6.8.

This type of loops is due to the fact that a transition with the empty word cannot be eliminated automatically by Unitex because it is associated with a transduction. Thus, the

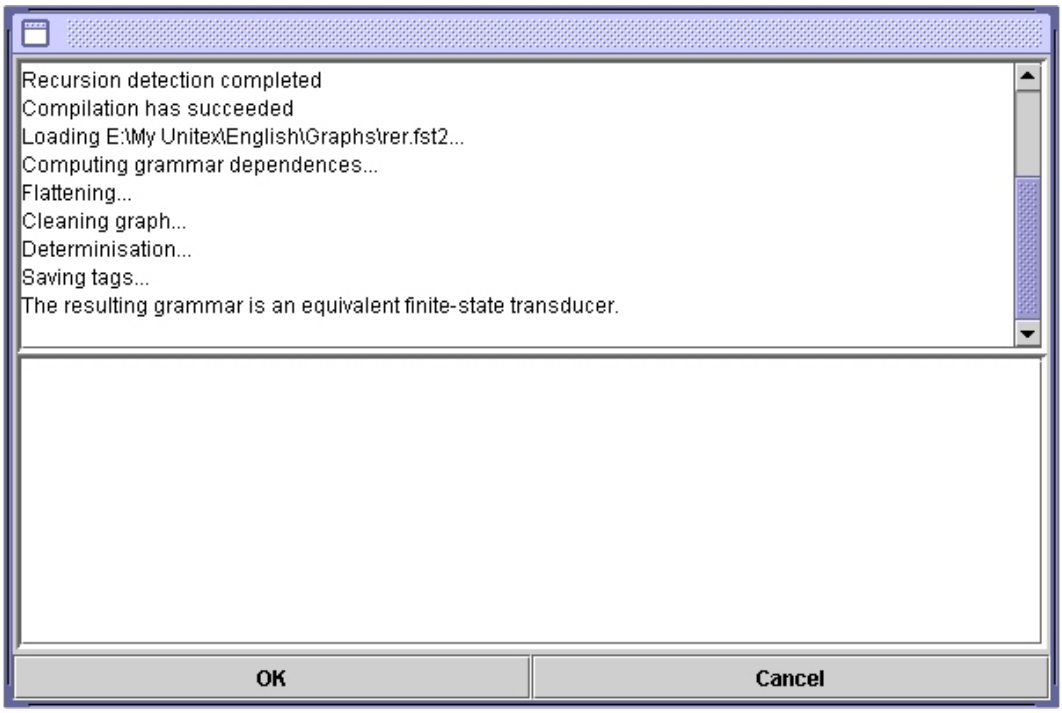

Figure 6.6: Resultat of the approximation of a grammar

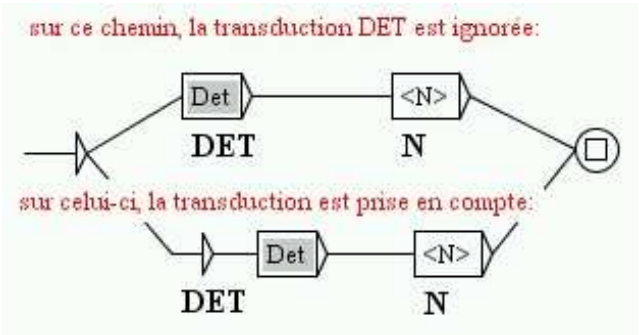

Figure 6.7: How to associate a transduction with a call to a subgraph

transition with the empty word of figure 6.8 will not be suppressed and will cause an infinite loop.

The second category of loop by epsilon concerns the call to subgraphs that can recognize the empty word. This case is illustrated in figure 6.9: if the subgraph `Adj` recognizes epsilon, there is an infinite loop that Unitex cannot detect.

The third possibility of infinite loops is related to recursive calls to subgraphs. Look at the graphs `Det` and `DetCompose` in figure 6.10. Each of these graphs can call the other

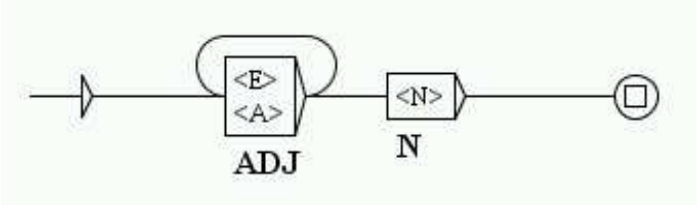

Figure 6.8: Infinite loop due to a transition by the empty word with a transductions

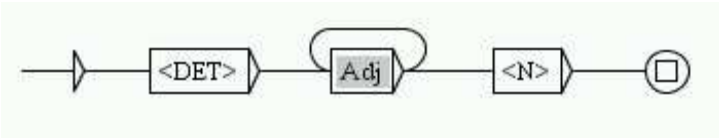

Figure 6.9: Infinite loop due to a call to a subgraph that recognizes epsilon

without reading any text. The fact that none of these two graphs has labels between the initial state and the call to the subgraph is crucial. In fact, if there were at least one label different from epsilon between the beginning of the graph `Det` and the call to `DetCompose`, this would mean that the Unitex programs exploring the graph `Det` would have to read the pattern described by that label in the text before calling `DetCompose` recursively. In this case the programs would loop infinitely only if they recognized the pattern an infinite number of times in the text, which is impossible.

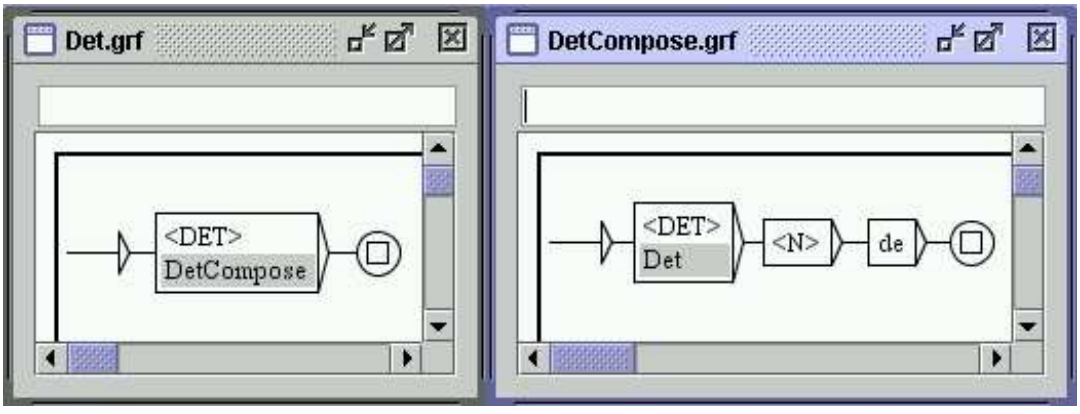

Figure 6.10: Infinite loop caused by two graphs calling each other

6.2.4 Error detection

In order to keep the programs from blocking or crashing, Unitex automatically detects errors during graph compilation. The graph compiler verifies that the principal graph does not recognize the empty word and searches for all possible forms of infinite loops. When an error is encountered an error message is displayed in the compilation window. Figure 6.11 shows the message that appears if one tries to compile the graph `Det` of figure 6.10.

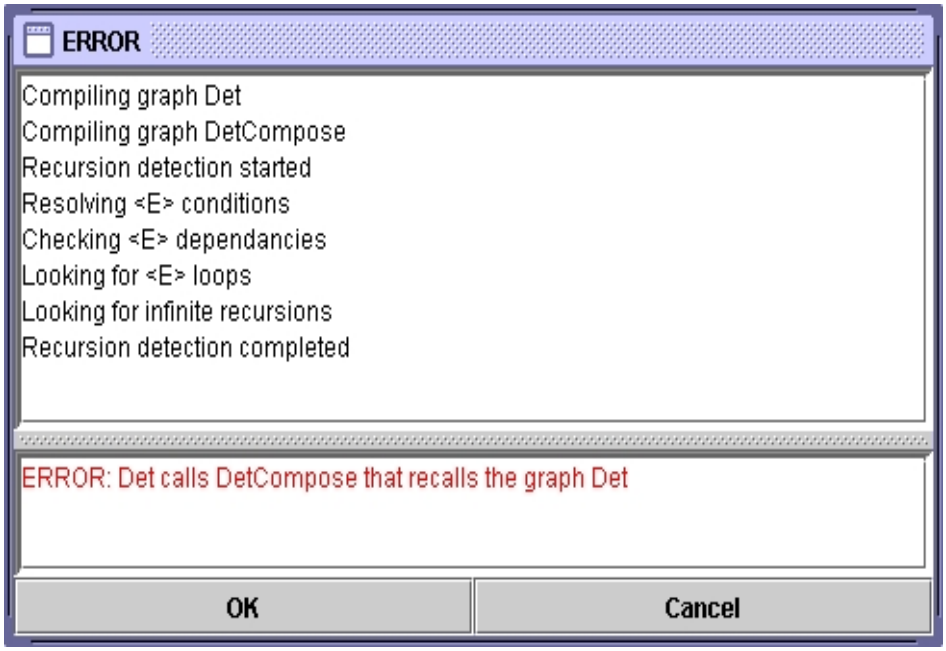

Figure 6.11: Error message when trying to compile `Det`

If you have started a pattern search be selecting a graph of the format `.grf` and Unitex discovers an error, the operation is automatically interrupted.

6.3 Exploring grammar paths

It is possible to generate the paths recognized by a grammar, for example to verify that it correctly generates the expected forms. For that, open the main graph of your grammar, and ensure that the graph window is the active window (the active window has a blue title bar, while the inactive windows have a gray title bar). Now go to the menu "FSGraph" and then to the "Tools" menu, and click on "Explore Graph paths". The Window of figure 6.12 appears.

The upper box contains the name of the main graph of the grammar to be explored. The following options relate to the outputs of the grammar:

- "Ignore outputs": the outputs are ignored;

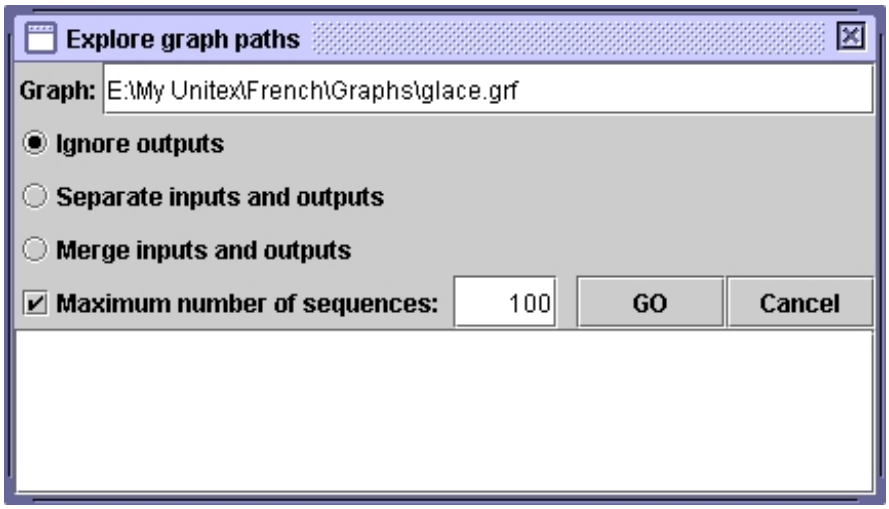

Figure 6.12: Exploring the paths of a grammar

- "Separate inputs and outputs": the outputs are displayed after the inputs (`a b c / A B C`);
- "merge inputs and outputs": Every output is posted immediately after the input to which it corresponds (`a/A b/B c/C`).

If the option "Maximum number of sequences" is activated, the specified number will be the maximum number of generated paths. If the option is not selected, all paths will be generated.

Here you see what is created for the graph in figure 6.13 with default settings (ignoring outputs, limit = 100 paths):

```
<NB> <boule> de glace à la pistache

<NB> <boule> de glace à la fraise

<NB> <boule> de glace à la vanille

<NB> <boule> de glace vanille

<NB> <boule> de glace fraise

<NB> <boule> de glace pistache

<NB> <boule> de pistache

<NB> <boule> de fraise
```

```
<NB> <boule> de vanille

glace à la pistache

glace à la fraise

glace à la vanille

glace vanille

glace fraise

glace pistache
```

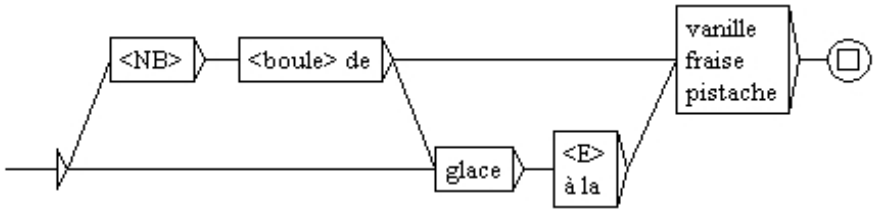

Figure 6.13: Sample graph

6.4 Graph Collections

It can happen that one wants to apply several grammars located in the same directory. For that, it is possible to automatically build a grammar starting from a tree structure of files. Let us suppose for example that one has the following tree structure:

- *Dicos*:
  - *Banque*:
    - `carte.grf`
  - *Nourriture*:
    - `eau.grf`
    - `pain.grf`
  - `truc.grf`

If one wants to gather all these grammars in only one, one can do it with the "Build Graph Collection" command in the "FSGraph Tools" sub-menu. One configures this operation by means of the window seen in figure 6.14.

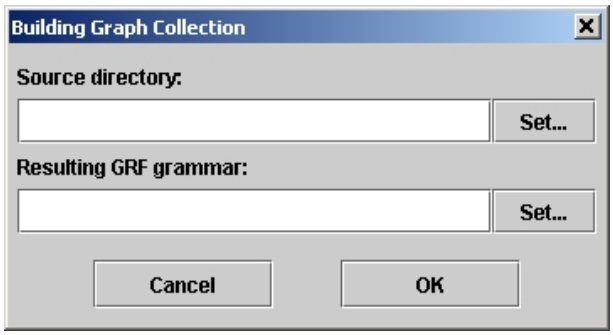

Figure 6.14: Building a Graph Collection

In the field "Source Directory", select the root directory which you want to explore (in our example, the directory *Dicos*). In the field "Resulting GRF grammar", enter the name of the produced grammar.

CAUTION: Do not place the output grammar in the tree structure which you want to explore, because in this case the program will try to read and to write simultaneously in this file, which will cause a crash.

When you click on "OK", the program will copy the graphs to the directory of the output grammar, and will create subgraphs corresponding to the various sub-directories, as one can see in figure 6.15, which shows the output graph generated for our example.

One can obsereve that one box contains the calls with subgraphs corresponding to sub-directories (here directories *Banque* and *Nourriture*), and that the other box calls all the graphs which were in the directory (here the graph `truc.grf`).

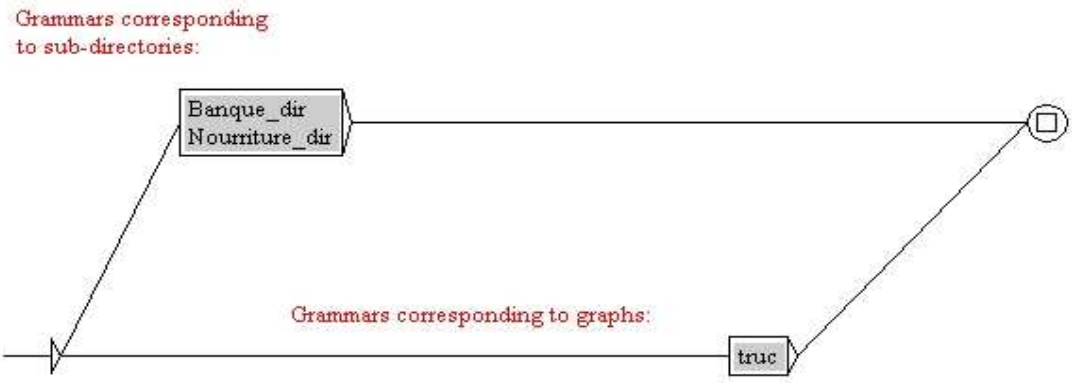

Figure 6.15: Main graph of a graph collection

6.5 Rules for applying transducers

This section describes the rules for the application of transducers along with the operaitons of preprocessing and the search for patterns. The following does not apply to inflection graphs and normalization graphs for ambiguous forms.

6.5.1 Insertion to the left of the matched pattern

When a transducer is applied in REPLACE mode, the output replaces the sequences that have been read in the text. In MERGE mode, the output is inserted to the left of the recognized sequences. Look at the transducer in figure 6.16.

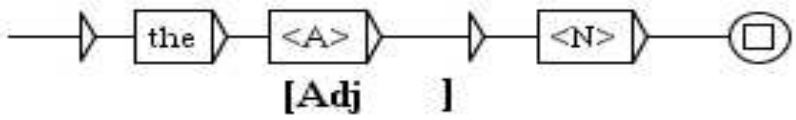

Figure 6.16: Example of a transducer

If this transducer is applied to the novel *Ivanhoe* by Sir Walter Scott in MERGE mode, the following concordance is obtained.

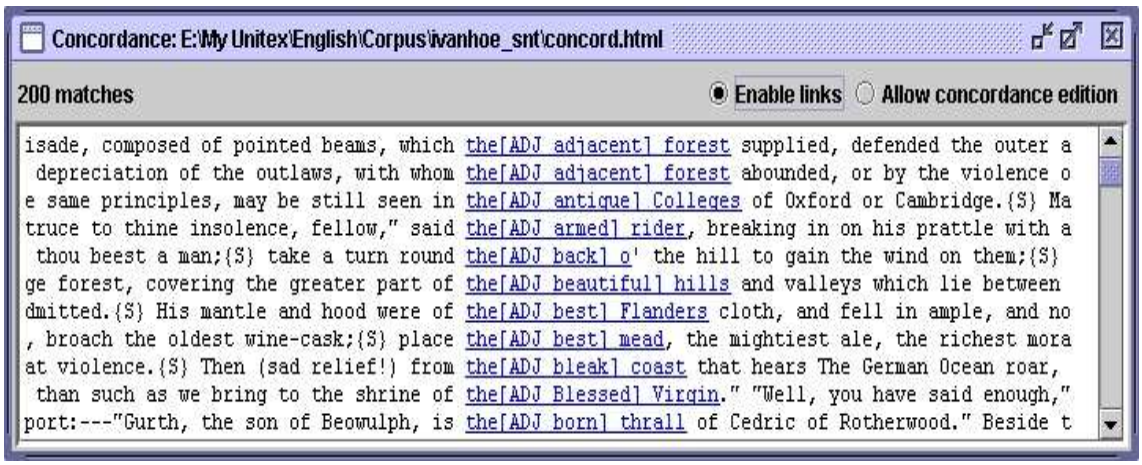

Figure 6.17: Concordance obtained in MERGE mode with the transducer of figure 6.16

6.5.2 Application while advancing through the text

During the preprocessing operations, the text is modified as it is being read. In order to avoid the risk of infinite loops, it is necessary that the sequences that are produced by a transducer will not be re-analyzed by the same one. Therefore, whenever a sequence is

inserted into the text, the application of the transducer is continued after that sequence. This rule only applies to preprocessing transducers, because during the application of syntactic graphs, the transductions do not modify the processed text but a concordance file which is different from the text.

6.5.3 Priority of the leftmost match

During the application of a local grammar, the collected occurrences are all indexed. During the construction of the concordance all these occurrences are presented (cf. figure 6.18).

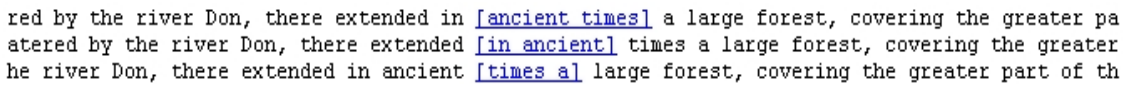

Figure 6.18: Occurrences are collected into concordance

On the other hand, if you modify a text instead of constructing a concordance, it is necessary to choose among these occurrences the one that will be taken into account. Unitex applies the following prioritisation rule for that purpose: the leftmost sequence is used.

If this rule is applied to the three occurrrences of the preceding concordance, the occurrence `[in ancient]` overlaps with `[ancient times]`. The first is retained because this is the leftmost occurrence and `[ancient times]` is eliminated. The following occurrence of `[times a]` is no longer in conflict with `[ancient times]` and can therefore appear in the result:

```
...Don, there extended [in ancient] [times a] large forest...
```

The rule of priority of the leftmost match is applied only when the text is modified, be it during preprocessing or after the application of a syntactic graph (cf. section 6.6.3).

6.5.4 Priority of the longest match

During the application of a syntactic graph it is possible to choose if the priority should be given to the shortest or the longest sequences or if all sequences should be retained. During preprocessing, the priority is always given to the longest sequences.

6.5.5 Transductions with variables

As we have seen in section 5.2.6, it is possible to use variables to store the text that has been analyzed by a grammar. These variables can be used in the preprocessing graphs and in the syntactic graphs.

You have to give names to the variables you use. These names can contain non-accentuated lower-case and upper-case letters between A and Z, digits and the character _ (underscore).

In order to define the end of the zone that is stored in a variable, you have to create a box that contains the name of the variable enclosed in the characters $ and (($ and) for the end of a variable). In order to use a variable in a transduction, its name must be preceded by the character $ (cf. figure 6.19).

Variables are global. This means that you can to define a variable in a graph and reference it in another as is illustrated in the graphs of figure 6.19:

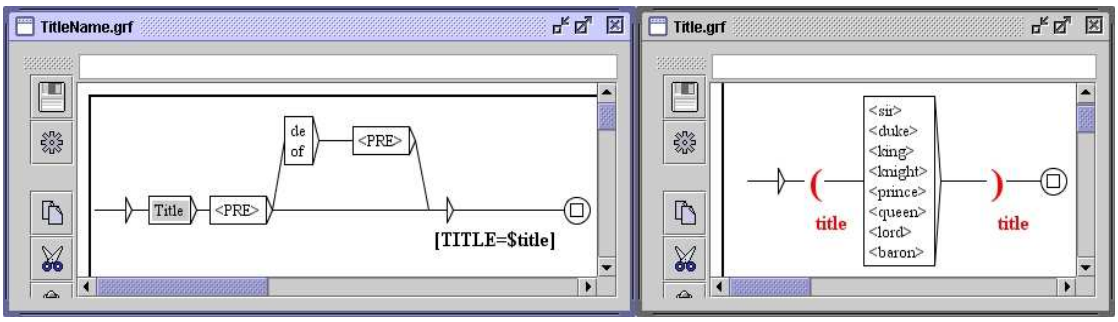

Figure 6.19: Definition of a variable in a subgraph

If the graph `TitleName` is applied in MERGE mode to the text *Ivanhoe*, the following concordance is obtained:

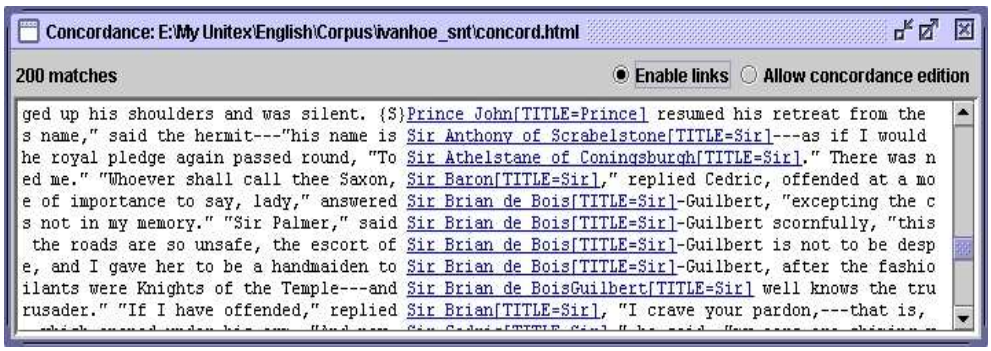

Figure 6.20: Concordance obtained by the application of the graph `TitleName`

Transductions with variables can be used to move groups of words. In fact, the application of a transducer in REPLACE mode inserts only the produced sequences into the text. In order to inverse two groups of words it is sufficient to store them into variables and produce a transduction with these variables in the desired order. Thus, the application of the transducer in figure 6.21 in REPLACE mode to the text *Ivanhoe* results in the concordance of figure 6.22.

The presence of a space to the right of each occurrence in the concordance of figure 6.22 is due to the insertion of a space after the `$NOUN $ADJ` in the transduction. Without this space, the result of the transduction would be glued to the right context (cf. figure 6.23).

In fact, the program `Locate` always considers the possibility of a facultative space between two boxes. In the present case the program tries to read a space between the box that

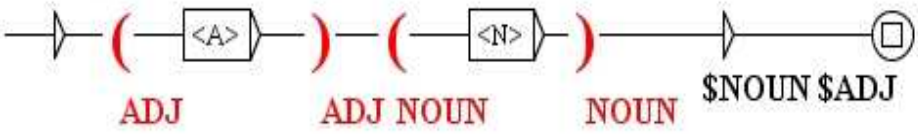

Figure 6.21: Inversion of words using two variables

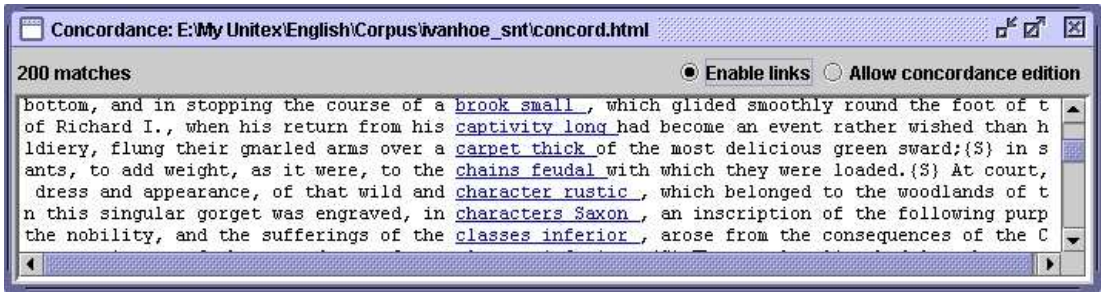

Figure 6.22: Result of the application of the transducer in figure 6.21

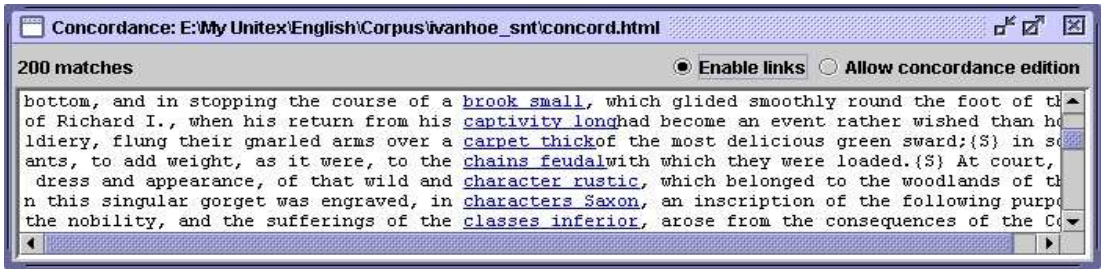

Figure 6.23: Spacing problem in REPLACE mode

constitutes the end of the variable `NOUN` and the box containing the transduction. If a space is read in REPLACE mode, it is erased because it is part of the text analyzed by the grammar. In order to avoid the loss of this space, it is therefore necessary to reinsert it by putting it into a transduction.

If the beginning or the end of variable is malformed (end of a variable before its beginning or absence of the beginning or end of a variable), it will be ignored during the transductions.

If you want to respect text spaces, the solution consists in making a difference between nouns that are followed by a space and other nouns. Figure 6.24 shows such a grammar. In the upper path, there is a space after `$NOUN $ADJ`. Applying this grammar in REPLACE mode builds the concordance shown on Figure 6.25. You can see in this concordance that previous spaces have been left unchanged and that no extra space was inserted. Note that

the boxes containing " " and # must immediately follow the <N> box. Placing them after the $NOUN) box would have no effect.

Figure 6.24: [[Caption missing]]

There is no limit of the number of possible variables.

The variables can be nested and even overlapping as is shown in figure 6.26:

Figure 6.25: [[Caption missing]]

6.6 Applying graphs to texts

This section only applies to syntactic graphs.

6.6.1 Configuration of the search

In order to apply a graph to a text, you open the text, then click on "Locate Pattern..." in the "Text" menu, or press <Ctrl+L>. You can then configure your search in the window shown in figure 6.27.

In the field "Locate pattern in the form of", choose "Graph" and select your graph by

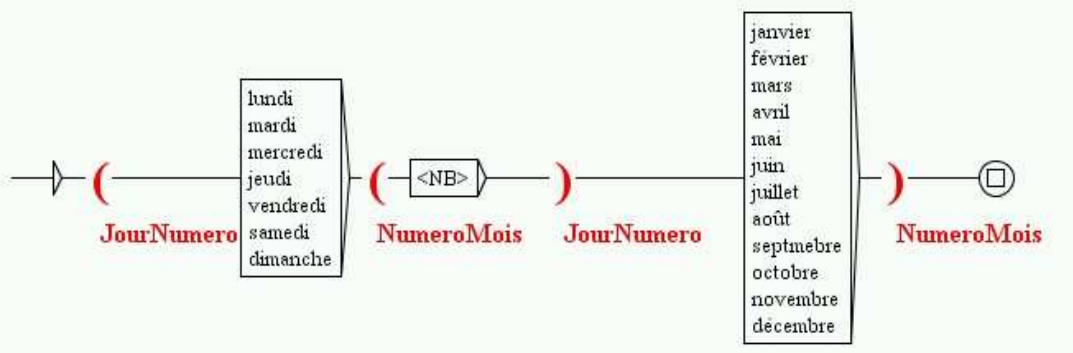

Figure 6.26: Nesting of variables

clicking on the "Set" button. You can choose a graph in `.grf` format (Unicode Graphs) or a compiled graph in `.fst2` format (Unicode Compiled Graphs). If your graph is in `.grf` format, Unitex will compile it automatically before starting the search.

The "Index" field allows to select the recognition mode.

- "Shortest matches" : give precedence to the shortest matches;
- "Longest matches" : give precedence to the longest sequences. This is the default mode;
- "All matches" : give out all recognized sequences.

The field "Search limitation" allows to limit the search to a certain number of occurrence. By default, the search is limited to the 200 first occurrences.

The field "Grammar outputs" concerns the use of the transductions. The mode "Merge with input text" allows to insert the sequences that are produced by the transductions. The mode "Replace recognized sequences" allows to replace the recognized sequences with the produced sequences. The third mode ignores all transductions. This latter mode is used by default.

After you have selected the parameters, click on "SEARCH" to start the search.

6.6.2 Concordance

The result of a search is an index file that contains the positions of all enountered occurrences. The window of figure 6.28 lets you choose whether to construct a concordance or modify the text.

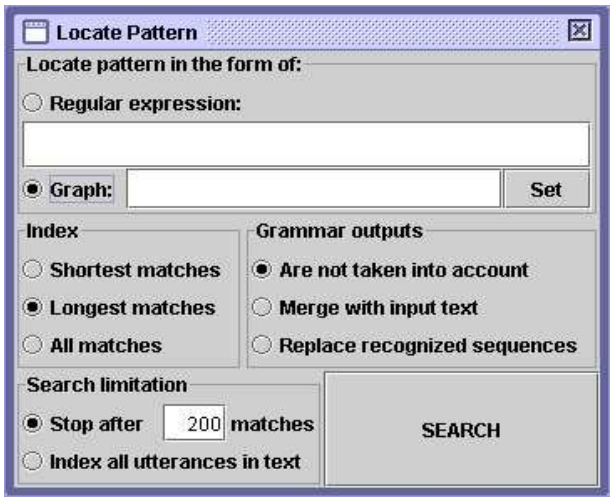

Figure 6.27: Window for pattern search

In order to display a concordance, you have to click on the botton "Build concordance". You can parameterize the size of left and right contexts in characters. You can also choose the sorting mode that will be applied to the lines of the concordance in the menu "Sort According to". For further details on the parameters of concordance construction, refer to the section 4.8.2

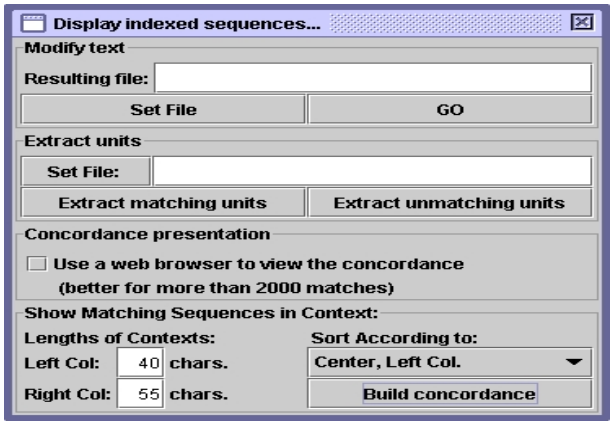

Figure 6.28: Configuration for displaying the encountered occurrences

The concordance is produced in the form of an HTML file. You can parameterize Unitex so that the concordances can be read using a web browser (cf. section 4.8.2).

If you display the concordances with the window provided by Unitex, you can access a recognized sequence in the text by clicking on the occurrence. If the text window is not

iconified and the text is not too long to be displayed, you see the selected sequence appear (cf. figure 6.29).

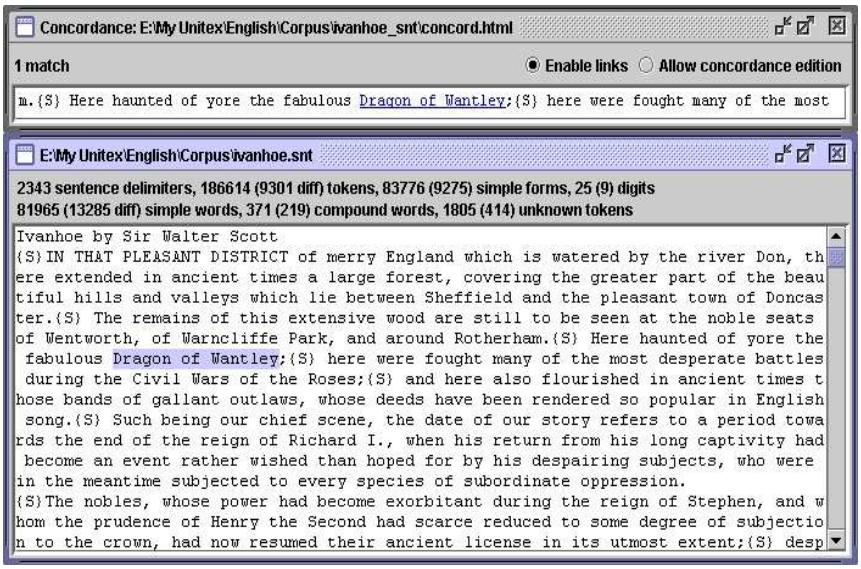

Figure 6.29: Selection of an occurrence in the text

Furthermore, if the text automaton has been constructed, and if the corresponding window is not iconified, clicking on an occurrence selects the automaton of the sentence that contains this occurrence.

6.6.3 Modification of the text

You can choose to modify the text instead of constructing a concordance. In order to do that choose a file name in the field "Modify text" in the window of figure 6.28. This file has to have the extension `.txt`.

If you want to modify the current text, you have to choose the corresponding `.txt` file. If you choose another file name, the current text will not be affected. Click on the "GO" button to start the modification of the text. The precedence rules that are applied during these operations are described in section 3.6.2.

After this operation the resulting file is a copy of the text in which all transductions have been taken into account. The normalization operations and the splitting into lexical units are automatically applied to this text file. The existing text dictionaries are not modified. Thus, if you have chosen to modify the current text, the modifications will be effective immediately. You can then start new searches on the text.

ATTENTION: if you have chosen to apply your graph ignoring the transductions, all occurrences will be erased from the text.

Chapter 7

Text automata

Natural languages contain lots of lexical ambiguities. The text automaton is an effective and visual means of representing these ambiguities. Each sentence of a text is represented by an automaton the paths of which express all possible interpretations.

This chapter presents the text automata, the details of their construction and the operations that can be applied. It is not possible at the moment to search for patterns on the text automaton nor to use rules in order to eliminate ambiguities.

7.1 Displaying text automata

The text automaton can express all possible lexical interpretations of the words. These different interpretations are the different entries presented in the dictionary of the text. Figure 7.1 shows the automaton of the fourth sentence of the text *Ivanhoe*.

You can see in figure 7.1 that the word `Here` has three interpretations here (adjective, adverb and noun), `haunted` two (adjective and verb), etc. All the possible combinations are expressed because each interpretation of each word is connected to all the interpretations of the following and preceding words.

In case of an overlap between a compound word and a sequence of simple words, the automaton contains a path that is labeled by the composite word, parallel to the paths that express the combinations of simple words. This is illustrated in figure 7.2, where the composite word `courts of law` is overlapping with a combination of simple words.

By construction, the automaton of the text doesn't contain any loops. One says that the text automaton is *acyclic*.

NOTE: the term text automaton is an abuse of the language. In fact, there is an automaton for each sentence of the text. Therefore, the combination of all these automata corresponds to the automaton of the text. Therefore the term text automaton is used even if this object is not really manipulated for practical reasons.

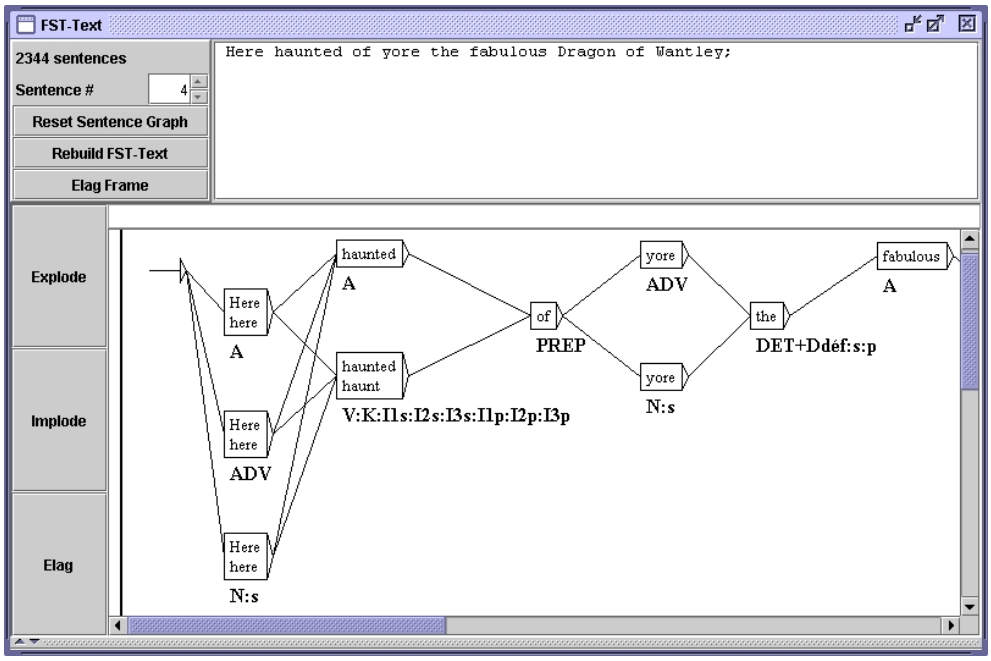

Figure 7.1: Example of the automaton of a sentence

7.2 Construction

In order to construct the text automaton, open the text, then click on "Construct FST-Text..." in the menu "Text". One should first split the text at sentence boundaries and apply the dictionaries. If sentence boundary detection is not applied, the construction program will split the text arbitrarily in sequences of 2000 lexical units instead of constructing one automaton per sentence. If the dictionaries are not applied, the sentence automaton that you obtain will consist of only one path made up of unknown words.

7.2.1 Construction Rules For Text Automata

The sentence automata are constructed starting from the text dictionaries. The obtained degree of ambiguity is therefore directly linked to the granularity of the descriptions of the used dictionaries. From the sentence automaton in figure 7.3, you can conclude that the word `which` has been coded twice as a determiner in two subcategories of the category `DET`. This granularity of descriptions will not be of any use if you are not interested in the grammatical category of this word. It is therefore necessary to adapt the granularity of the dictionaries to the intended use.

For each lexical unit of the sentence, Unitex searches for all possible interpretations in the dictionary of the simple words of the text. Afterwards all lexical units that have an interpretation in the dictionary of the composite words of the text are sought. All the combinations of their interpretations constitute the sentence automaton.

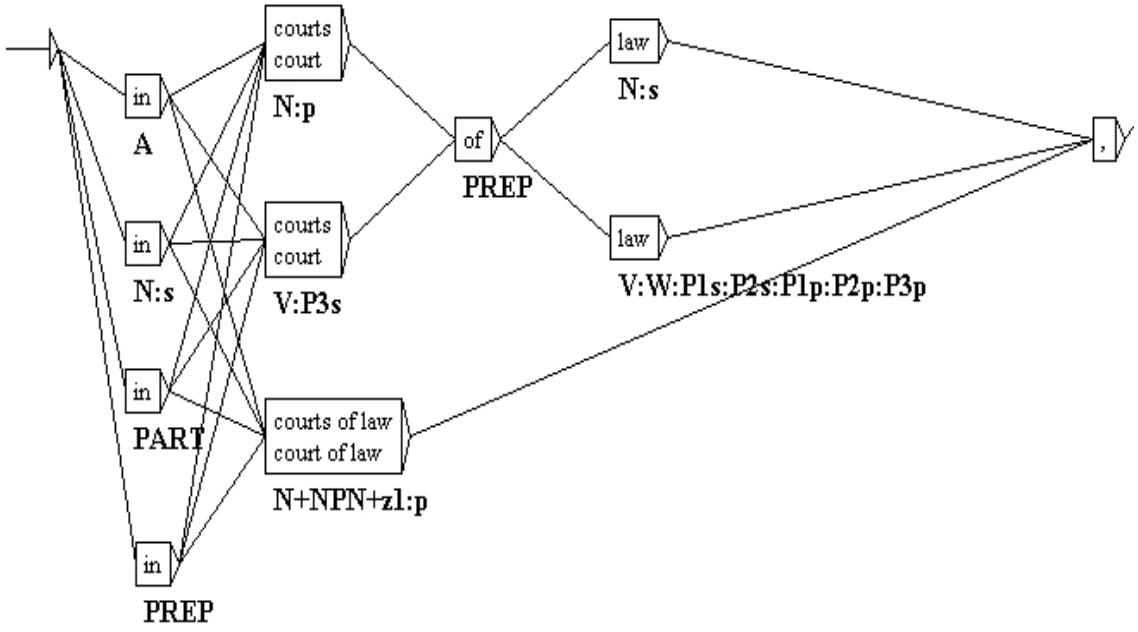

Figure 7.2: Overlap between a compound word and a combination of simple words.

NOTE: If the text contains lexical labels (*e.g.* `{out of date,.A+z1}`), these labels are reproduced identically in the automaton without trying to decompose the sequences which they represent.

In each box, the first line contains the inflected form found in the text, and the second line contains the canonical form if it is different. The other information is coded below the box. (cf. section 7.4.1).

The spaces that separate the lexical units are not copied into the automaton save the spaces inside composite words.

The case of lexical units is conserved. For example, if the word `Here` is encountered, the capital letter is preserved (cf. figure 7.1). This choice allows to keep this information during the transition to the text automaton, which could be useful for applications where case is important such as recognition of proper names.

7.2.2 Normalization of ambiguous forms

During construction of the automaton, it is possible to effect a normalization of ambiguous forms by applying a normalization grammar. This grammar has to be called `Norm.fst2` and must be placed in your personal folder, in the subfolder `/Graphs/Normalization` of the desired language. The normalization grammars for ambiguous forms are described in section 6.1.3.

If a sequence of the text is recognized by the normalization grammar, all the interpretations that are described by the grammar are inserted into the text automaton. Figure 7.4

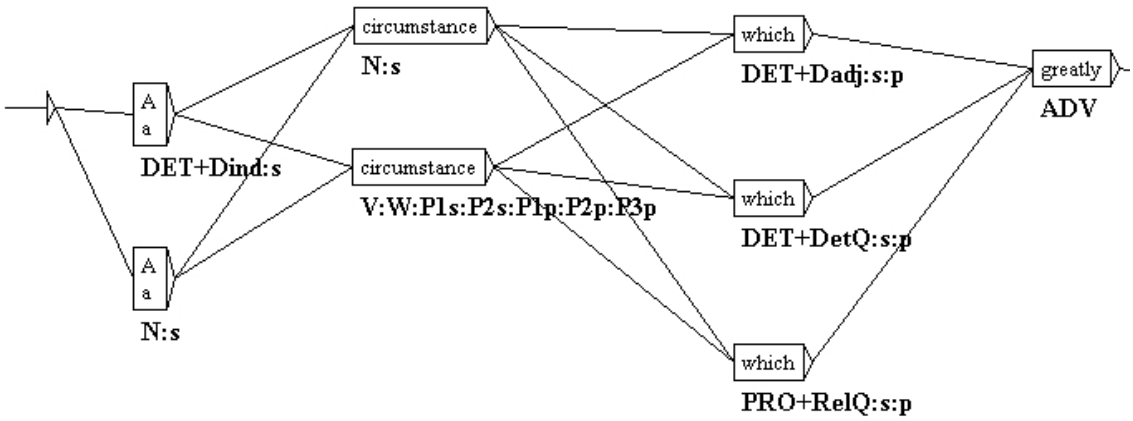

Figure 7.3: Double entry for `which` as a determinator

shows the extract of the grammar used for French that makes the ambiguity of the sequence `l'` explicit.

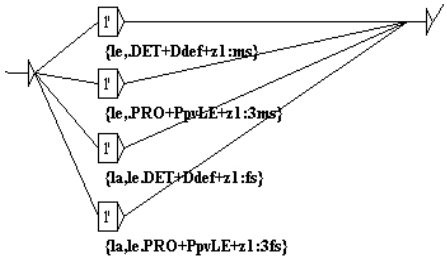

Figure 7.4: Normalization of the sequence `l'`

If this grammar is applied to a French sentence containing the sequence `l'`, a sentence automaton that is similar to the one in figure 7.5 is obtained.

You can see that the four rules for rewriting the sequence `l'` have been applied, which has added four labels to the automaton. These labels are concurrent with the two preexisting paths for the sequence `l'`. The normalization at the time of the construction of the automaton allows to add paths to the automaton but not to erase paths. When the disambiguation functionality will be available, it will allow to eliminate the paths that have become superfluous.

7.2.3 Normalization of clitical pronouns in Portuguese

In Portuguese the verbs in the future tense and in the conditional can be modified by the insertion of one or two clitical pronouns between the root and the suffix of the verb. For example, the sequence *dir-me-ão* (*they will tell me*), corresponds to the complete verbal form *dirão*, associated with the pronoun *me*. In order to be able to manipulate this rewritten form,

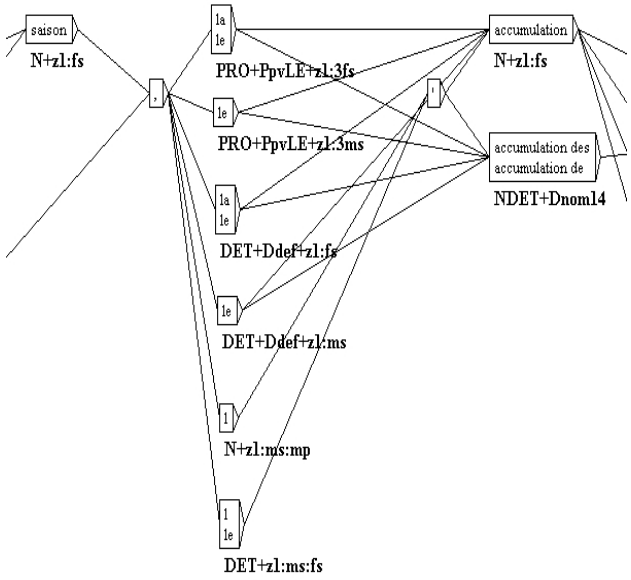

Figure 7.5: Automaton that has been normalized with the grammar of figure 7.4

it is necessary to introduce it into the text automaton in parallel to the original form. Thus, the user could search one or the other form. The figures 7.6 and 7.7 show the automaton of a sentence after the normalization of the clitics.

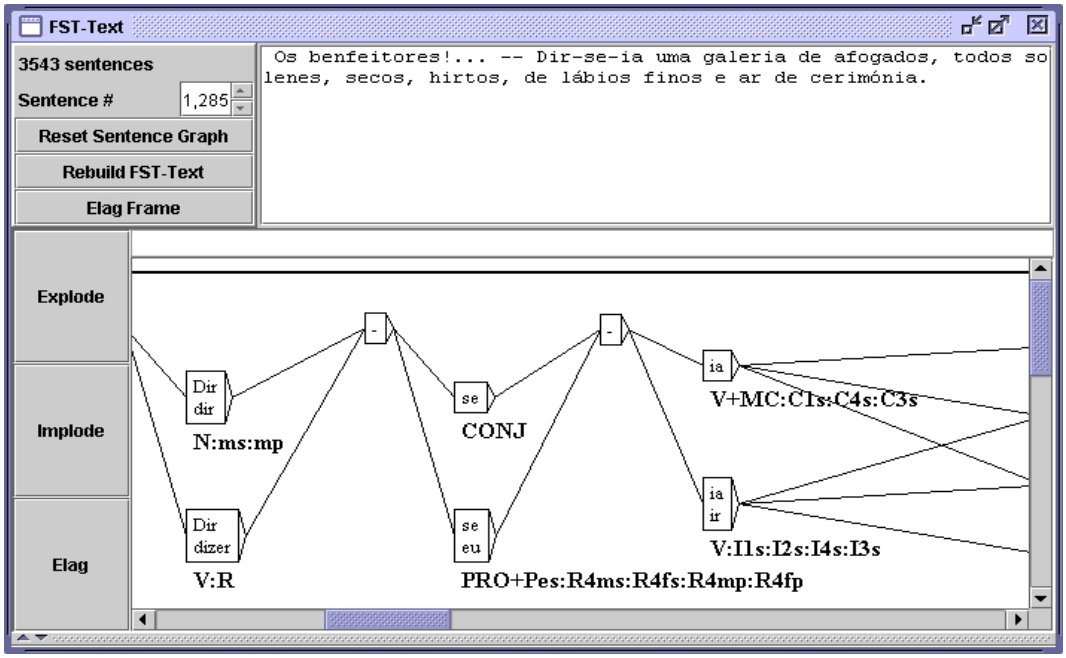

Figure 7.6: Non-normalized phrase automaton

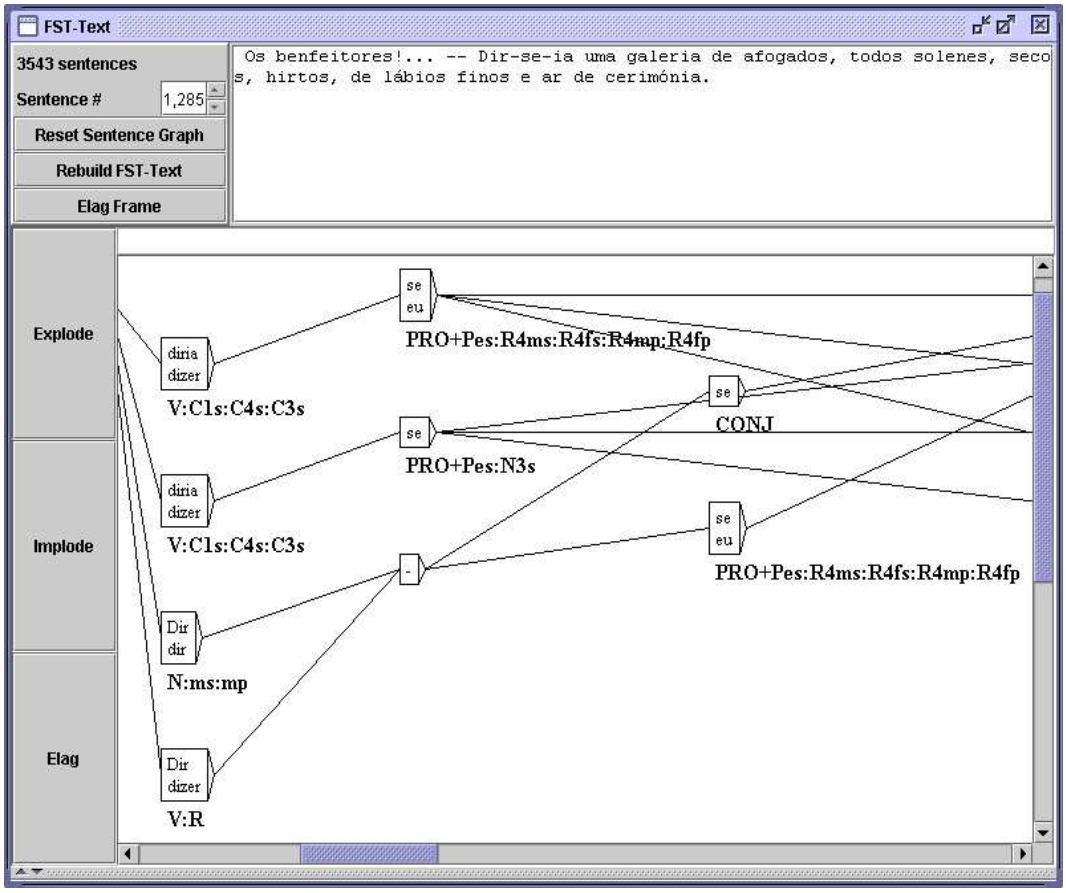

Figure 7.7: Normalized phrase automaton

The program `Reconstrucao` allows to construct a normalization grammar for these forms for each text dynamically. The thus produced grammar can then be used for normalizing the text automaton. The configuration window of the automaton construction suggests an option "Build clitic normalization grammar" (cf. figure 7.10). This option automatically starts the construction of the normalization grammar, which is then used to construct the text automaton, if you have selected the option "Apply the Normalization grammar".

7.2.4 Keeping the best paths

It can possible that an unknown word jeopardizesXXX the text automaton by overlapping with a completely labeled sequence. Thus, in the automaton of figure 7.8, it can be seen that the adverb `aujourd'hui` overlaps with the unknown word `aujourd`, followed by an apostrophe and the past participle of the verb `huir`.

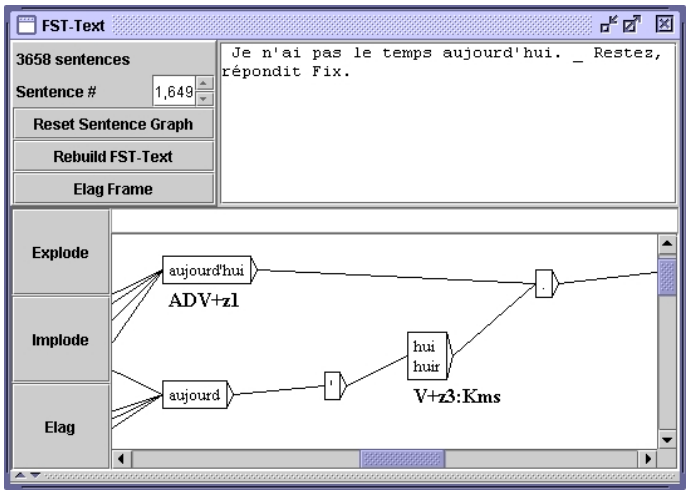

Figure 7.8: Ambiguity due to a sentence conaining an unkown word

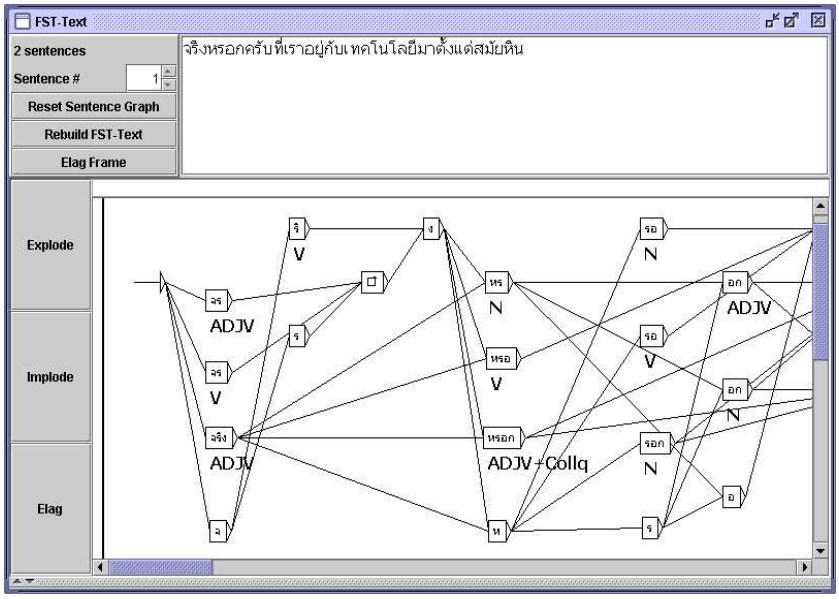

Figure 7.9: Automaton of a thaï sentence

This phenomenon can also be found in the treatment of certain Asian languages like Thaï. When the words are not delimited, there is no other solution than to consider all possible combinations, which causes the creation of numerous paths carrying unknown words that are mixed with the labeled paths. Figure 7.9 shows an example of such an automaton of a Thaï sentence.

It is possible to suppress parasite paths. You have to select the option "Clean Text FST" in the configuration window for the construction of the text automaton (cf. figure 7.10).

This option indicates to the automaton construction program that it should clean up each sentence automaton.

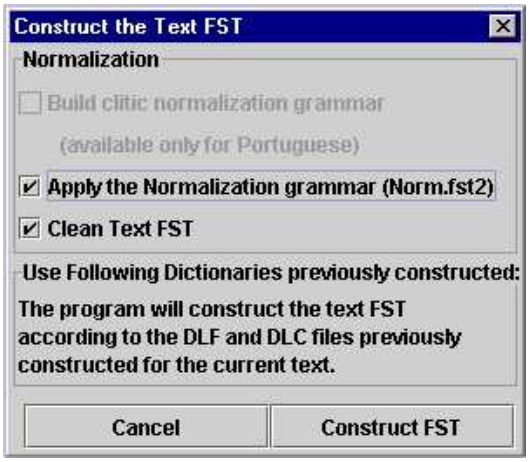

Figure 7.10: Configuration of the construction of the text automaton

This cleaning is carried out according to the following principle: if several paths are concurrent in the automaton, the program keeps those that contain the fewest unknown words.

Figure 7.11 shows the automaton of figure 7.9 after cleaning.

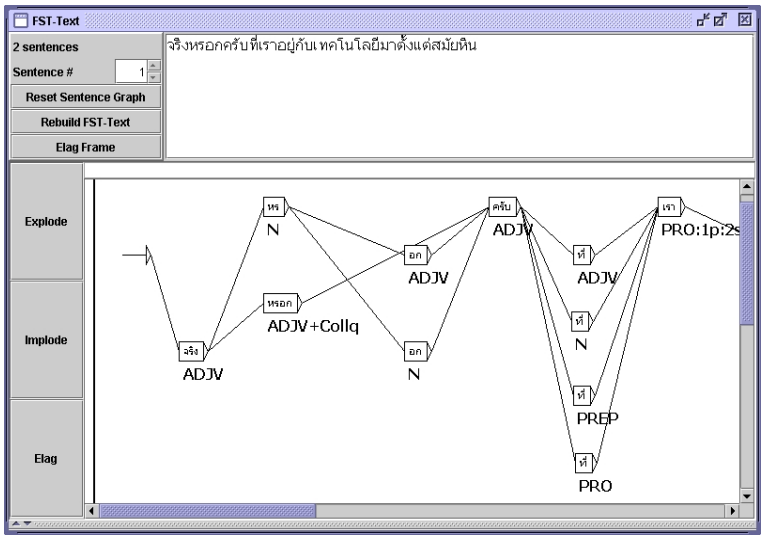

Figure 7.11: Automaton of figure 7.9 after cleaning

7.3 Resolving Lexical Ambiguities with ELAG

The ELAG program allows for applying grammars for ambiguity removal to the text automaton.

This powerful mechanism allows for everybody to write rules on his own independent from already existing rules.

This chapter shortly presents the grammar formalism used by ELAG and describes how the program works.

For more details, the reader may refer to [**?**] and [**?**].

7.3.1 Grammars For Resolving Ambiguities

The grammars used by ELAG have a special syntax. They consist of two parts which we call if and $then$ parts.

The if part of an ELAG grammar is divided in two parts which are divided by a box containing the <!>.

The $then$ part is divided the same way using the <=> symbol. The meaning of a grammar is like the following:

In the text automaton, if a section of the if part is recognized, then it must also be recognized by

the "then" part of the grammar, or it will be withdrawn from the text automaton.

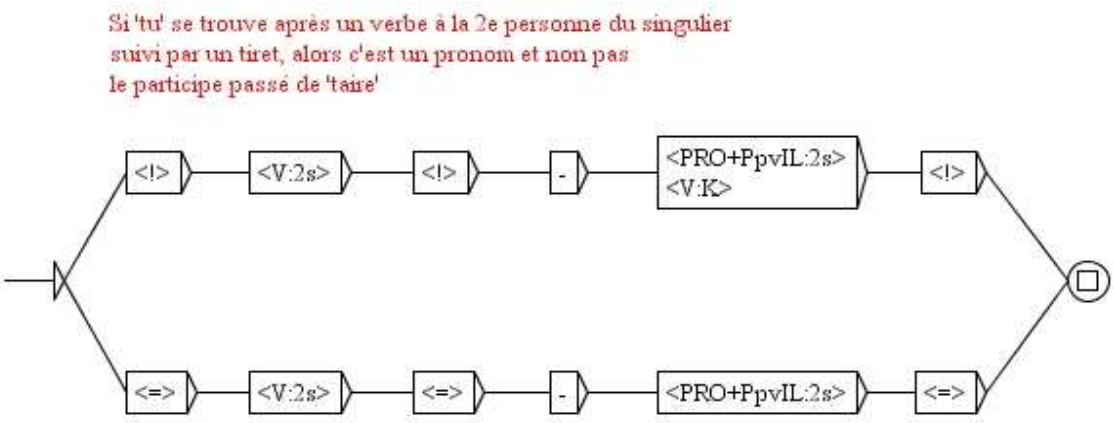

Figure 7.12: Exemple de grammaire ELAG

Figure 7.12 shows an example of a grammar. The if part recognizes a verb in the

2nd person singular followed by a dash and tu, either as a pronoun, or as a past participle of the verb $taire$.

The $then$ part imposes that tu is then regarded as a pronoun. Figure 7.13 shows the

result of the application of this grammar on the sentence "*Feras-tu cela bientôt ?*". One can see in the automaton

at the bottom that the path corresponding to tu past participle was eliminated.

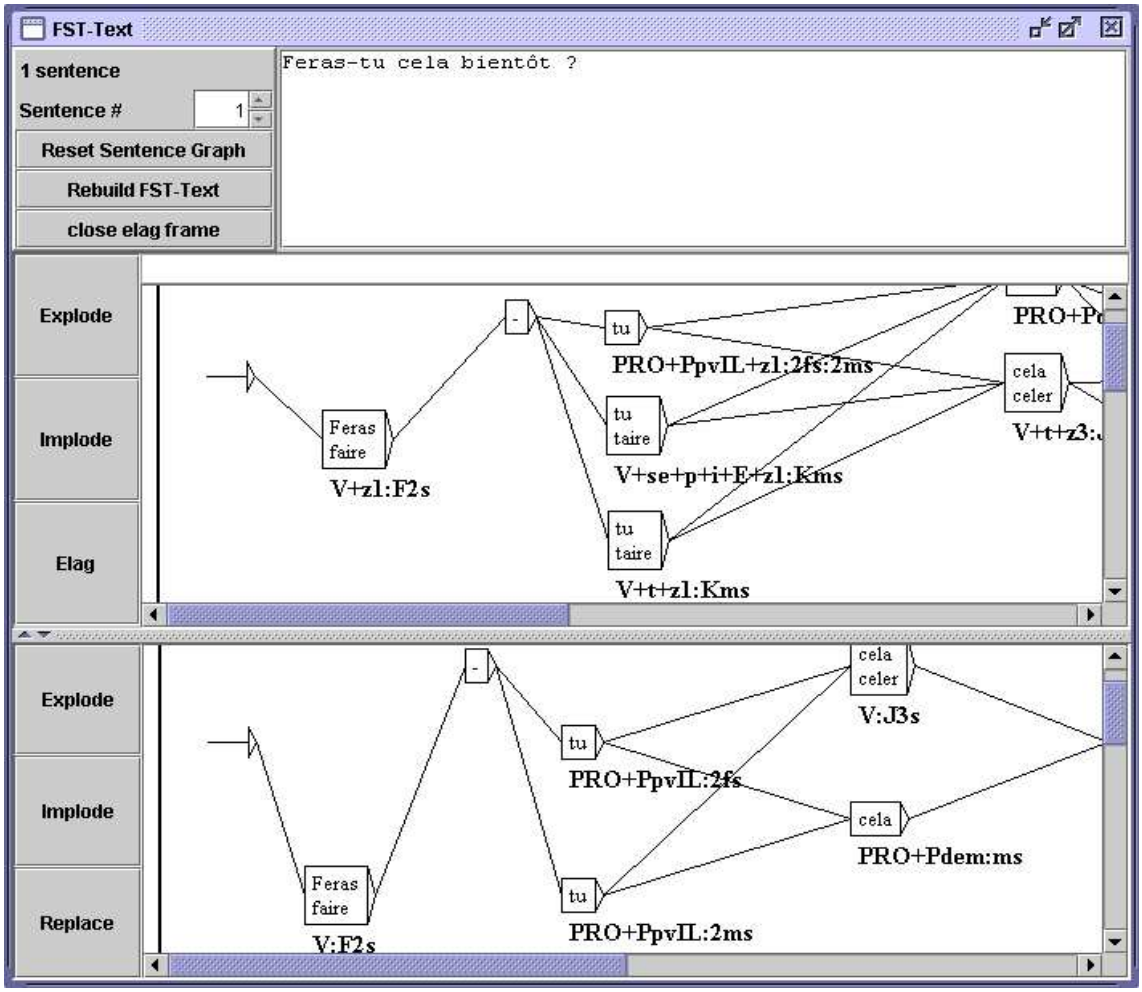

Figure 7.13: Résultat de l'application de la grammaire de la figure 7.12

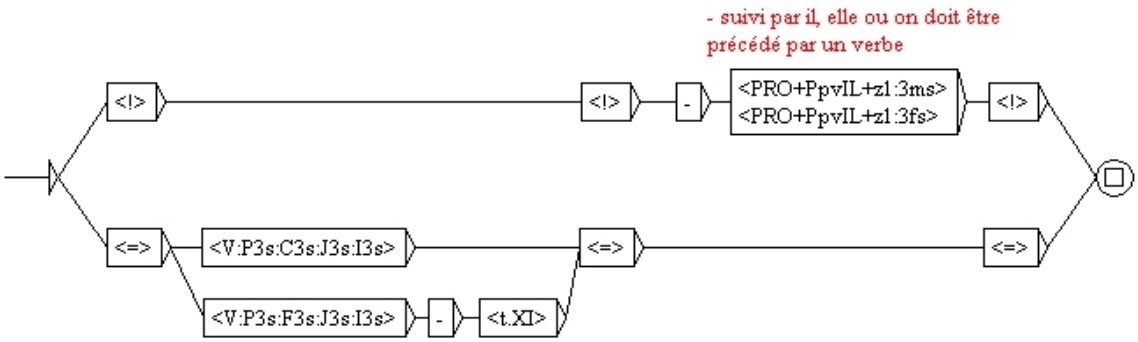

Figure 7.14: Utilisation du point de synchronisation

Point of synchronization

The if and $then$ parts of an ELAG grammar are divided into two parts by the <!> in the if part, and <=> in the $then$ part. These symbols form a $point of synchronization$. This makes it possible to write rules in which the if and $then$ constraints are not necessarily aligned, as it is for example the case in figure 7.14. This grammar is interpreted in the following way: if a dash is found followed by il, $elle$ or on, then this dash must be preceded by a verb, possibly followed by $-t$. So, if one considers the sentence of the figure 7.15 beggining with *Est-il*, one can see that all non-verb interpretations of Est were removed.

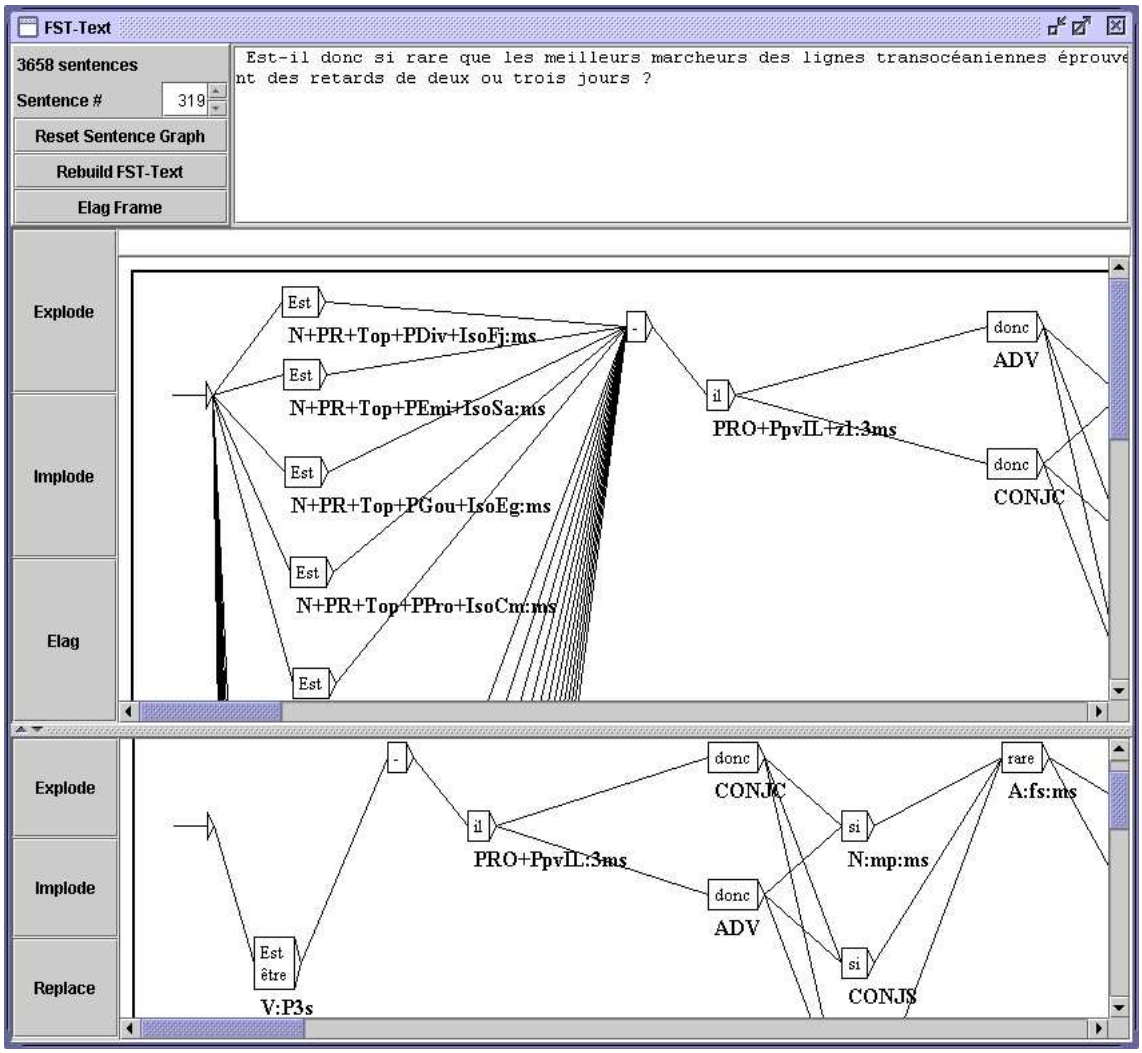

Figure 7.15: Résultat de l'application de la grammaire de la figure 7.14

7.3.2 Compiling ELAG Grammars

Before being able to be applied to a text automaton, an ELAG grammar must be compiled in a `.rul` file. This operation is carried out via the "Elag Rules" command in the "Text" menu, which opens the windows shown in figure 7.16.

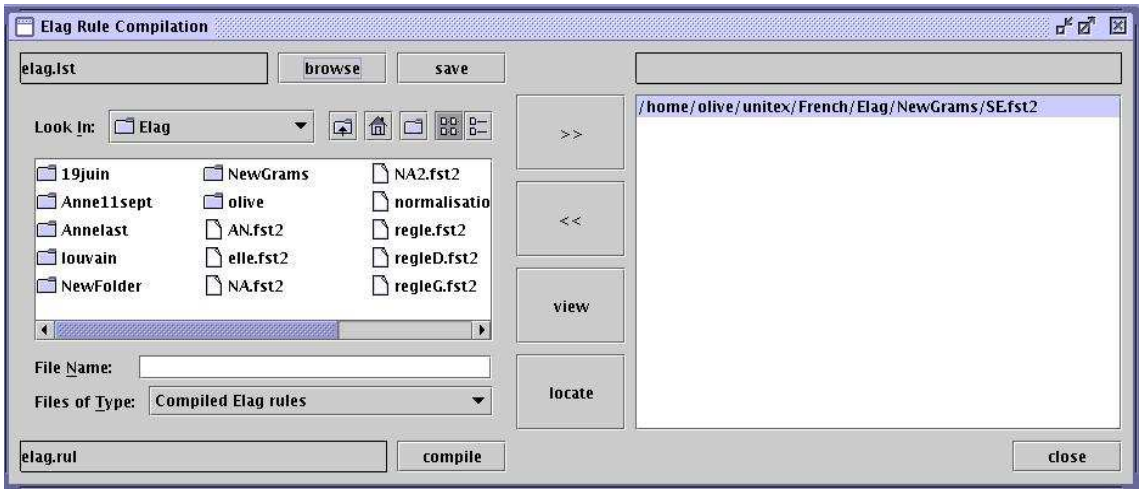

Figure 7.16: Fenêtre de compilation des grammaires ELAG

If the frame on the right already contains grammars which you don't wish to use, you can withdraw them with the << button. Then select your grammar in the file explorer located in the left frame, and click on the >> button to add it to the list in the right frame. Then click on the *compile* button. This will launch the program `ElagComp` which will compile the selected grammar and create a file named *elag.rul*.

If you selected your grammar in the right frame, you can search patterns which it recognizes by clicking on the *locate* button. This opens the window "Locate Pattern" and automatically enters a graph name ending with `-conc.fst2`. This graphs corresponds to the *if* part of the grammar. You can thus obtain the occurrences of the text to which the grammar will apply.

NOTE: The `-conc.fst2` file used to locate the *then* part of a grammar is generated at the time when ELAG grammars are compiled by means of the *compile* button. It is thus necessary initially to have your grammar compiled before searching using the *locate* button.

7.3.3 Resolving Ambiguities

Once you have compiled your grammar into an `elag.rul` file, you can apply it to a text automaton. In the text automaton window, click on the *elag* button. A dialog box will appear which asks for the the `.rul` file to use (see figure 7.17). As the default file is likely to be `elag.rul`, simply click on "OK". This will launch the `Elag` which will resolve the ambiguities.

Once the program has finished you can view the resulting automaton by clicking on the *Elag Frame* button. As you can

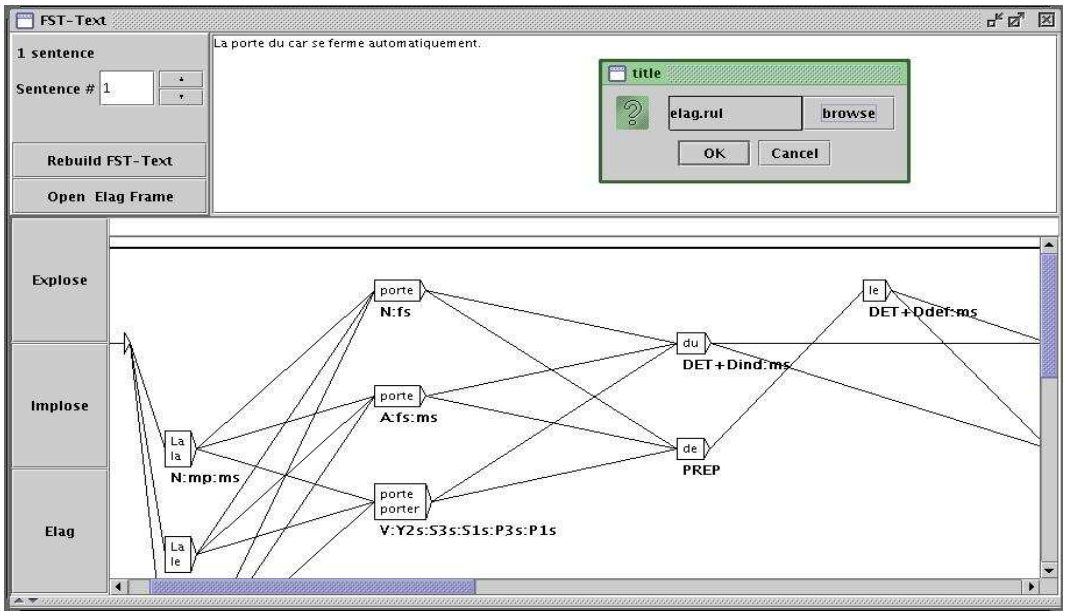

Figure 7.17: Fenêtre de l'automate du texte

see in figure 7.18, the windows is separated into two parts: The original text automaton can be seen on the top,

and the result at the bottom.

Don't be surprised if the automaton shown at the bottom seems more complicated. This results from the fact that factorized lexical entries[1] were exploded in order to treat each inflectional interpretation separately. To refactorize these entries, click on the *implode* button. Clicking on the button *explode* shows you an exploded view of the text automaton. If you click on the *replace* button, the resulting automaton will become the new text automaton. Thus, if you use other grammars, they will apply to the already partially disambiguated automaton, which makes it possible to accumulate the effects of several grammars.

7.3.4 Grammar collections

It is possible to gather several ELAG grammars into a grammar collection, in order to apply them in one step. The sets of ELAG grammars are described in `.lst` files. They are managed through the window for compiling ELAG grammars (figure 7.16). The label on the top left indicates the name of the current collection, by default `elag.lst`. The contents of this collection are displayed in the right part of the window.

To modify the name of the collection, click on the *browse* button. In the dialog box that appears, enter the `.lst` file name for the collection.

To add a grammar to the collection, select it in the file explorer in the left frame, and click on the >> button. Once you selected all your grammars, compile them by clicking on the

[1] Entries which gather several different inflectional interpretations, such as for example: `{se,.PRO+PpvLE:3ms:3fs:3mp:3fp}`.

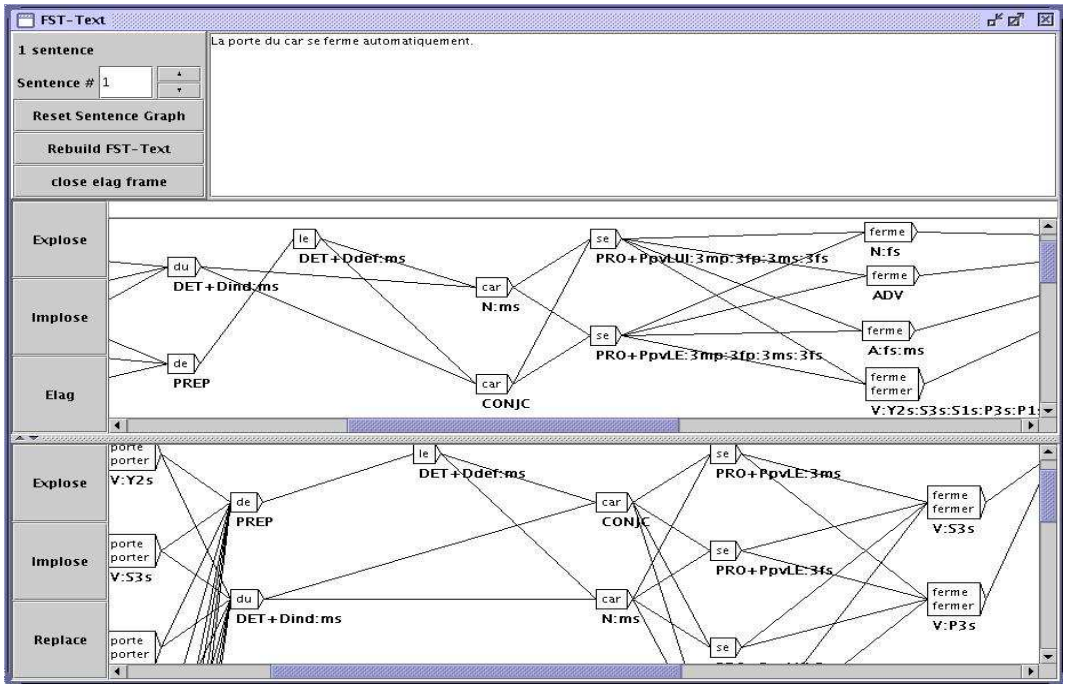

Figure 7.18: Fenêtre de l'automate du texte séparée en deux

compile button. This will create a `.rul` file bearing the name indicated at the bottom right (the name of the file is obtained by replacing the `.lst` by the `.rul` extension).

You can now apply your grammar collection. As explained above, click on the *elag* button in the text automaton window. When the dialog asks for the `.rul` file to use, click on the *browse* button and select your collection. The resulting automaton is identical to that which would have been obtained by applying each grammar successively.

7.3.5 Window For ELAG Processing

At the time of disambiguation, the `Elag` program is launched in a processing window which makes it possible to see the messages printed by the program during its execution.

For example, when the text automaton contains symbols which do not correspond to the set of ELAG labels (see the following section), a message indicates the nature of the error. In the same way, when a sentence is rejected (all possible analyses were eliminated by grammars), a message indicates the number of the sentence. That makes it possible to locate the source of the problems quickly.

Evaluation of ambiguity removal

The evaluation of the rate of ambiguity is not based solely on the average numebr of interpretations per word. In order to get a more representative measure, the system also takes into account the various combinations of words. While ambiguities are resolved, the `elag`

program calculates the number of possible analyses in the text automaton before and after the modification (which corresponds to the number of possible paths through the automaton). Based on this value, the program computes the average ambiguity by sentence and word. It is this last measure which is used to represent the rate of ambiguities of the text, because it does not vary with the size of the corpus, nor with the number of sentences within. The formula applied is:

$$\textit{taux d'ambiguïtés} = exp^{\frac{log(number-of-paths)}{text-length}}$$

The relationship between the rate of ambiguities before and after applying the grammars gives a measure of their efficiency.

All this information is displayed in the ELAG processing window.

7.3.6 Description Of The Tag Sets

The `Elag` and `ElagComp` require a formal description of the set of labels of the dictionaries used. This description consists roughly speaking of an enumeration of all the grammatical categories present in the dictionaries, with, for each one of it, the syntactic and inflectional code list which are associated for them, and a description of their possible combinations. This information is described in the file named `tagset.def`.

`tagset.def` file

Here is an extract of the `tagset.def` file used for French.

```
NAME français


POS ADV

.


POS PRO

inflex:

  pers   = 1 2 3
```

```
  genre  = m f

  nombre = s p

discr:

  subcat = Pind Pdem PpvIL PpvLUI PpvLE Ton PpvPR PronQ Dnom Pposs1s...

complete:

Pind     <genre> <nombre>

Pdem     <genre> <nombre>

Pposs1s  <genre> <nombre>

Pposs1p  <genre> <nombre>

Pposs2s  <genre> <nombre>

Pposs2p  <genre> <nombre>

Pposs3s  <genre> <nombre>

Pposs3p  <genre> <nombre>

PpvIL    <genre> <nombre> <pers>

PpvLE    <genre> <nombre> <pers>

PpvLUI   <genre> <nombre> <pers>      #

Ton      <genre> <nombre> <pers>      # lui, elle, moi

PpvPR                                 # en y

PronQ                                 # où qui que quoi

Dnom                                  # rien

.


POS A ## adjectifs
```

```
inflex:
  genre  = m f
  nombre = s p
cat:
  gauche = g
  droite = d
complete:
<genre> <nombre>
_  # pour {de bonne humeur,.A}, {au bord des larmes,.A} par exemple
.


POS V
inflex:
  temps  = C F I J K P S T W Y G X
  pers   = 1 2 3
  genre  = m f
  nombre = s p
complete:
W
G
C <pers> <nombre>
```

```
F <pers> <nombre>

I <pers> <nombre>

J <pers> <nombre>

P <pers> <nombre>

S <pers> <nombre>

T <pers> <nombre>

X 1 s   # eussé dussé puissé fussé (-je)

Y 1 p

Y 2 <nombre>

K <genre> <nombre>

.
```

The # symbol indicates that the remainder of the line is a comment.

A comment can appear at any place in the file. The file always starts with the word `NAME`, followed

by an identifier (`français`, for example). This is followed by the sections `POS` (for Part-Of-Speech),

for each grammatical category. Each section describes the structure of the labels of the lexical entries

belonging to the grammatical category concerned. Each section is composed of 4 parts which are all optional:

- `inflex`: this part enumerates the inflectional codes belonging to the grammatical category.

  For example, the codes `1,2,3` which indicate the person of the entry are relevant for pronouns but not for adjectives.

  Each line describes an inflexional attribute (gender, time, etc.) and is made up of the attribute name, followed by the `=`

  character and the values which it can take; For example, the following line declares an attribute $pers$ being able to

  take the values 1, 2 or 3:

```
pers = 1 2 3
```

- `cat`: this part declares the syntactic and semantic attributes which can be allotted to the

  entries belonging to the grammatical category concerned. Each line describes an attribute and the values which it can take.

  The codes declared for the same attribute must be exclusive. In other words, an entry cannot take more than one value for the

  same attribute.

  On the other hand, labels can exist which don't take any value for a given attribute. For example, to define the attribute

  `niveau_de_langue` which can take the values `z1`, `z2` and `z3`, the following line can be written:

```
niveau_de_langue = z1 z2 z3
```

- `discr`:

  this part consists of a declaration of a unique attribute. The syntax is the same as in the `cat` part and the attribute

  described here must not be repeated.

  This part allows for dividing the grammatical category in *discriminating* sub categories in which the entries

  have similar inflectional attributes.

  For pronouns for example, a person indicator is assigned to entries that are part of the personal pronoun sub category

  but not to relative pronouns. These dependencies are described in the `complete` part;

- `complete`:

  In this part the morphological tags of the words in the current grammatical category are described.

  Each line describes a valid combination of inflectional codes by their discriminating sub category (if

  such a category was declared). If an attribute name is specified in angle brackets (< and >),

  this siginifies that any value of this attribute might occur.

  It is possible as well to declare that an entry does not take any inflexional feature by means of a line

containing only the character _ (underscore.. So for example, if we consider
that the following lines extracted from the section describing the verbs:

```
W

K <genre> <nombre>
```

They make it possible to declare that verbs in base form (indicated by the code W) do not have
other inflectional features while the forms in past participle (code K) are also alotted
with a gender and a number.

Description of the inflextional codes

The principal function of the `discr` part is to divide the labels into subcategories having
similar morphological behavior. These subcategories are then used to falititate writing the `complete` part.
For the legibility of the ELAG grammars, it is desirable that the elements of the same subcategory all have
the same inflectional behavior; in this case the `complete` part is made up of only one line per subcategory.

Let us consider for example the following lines, extracts of the pronoun description:

```
Pdem  <genre> <nombre>

PpvIl <genre> <nombre> <pers>

PpvPr
```

These lines mean:

- all the demonstrative pronouns (`PRO+Pdem>`) have indication of gender and number, and any other;
- personal pronouns in nominative (`<PRO+PpvIl>`) are labelled morphologically by person, gender and number;

- the prepositional pronouns (*en*, *y*) do not have any inflectional feature.

All combinations of inflectional features and discriminants which appear in the dictioanries must be described
in the file `tagset.def`, or else the corresponding entries will be restrected
by ELAG.
If words of the same subcategory differ by their inflectional features, it is necessary to write several lines into the
`complete` part. The disadvantage of this method of description is that it becomes difficult to make the
distinction between such words in an ELAG grammar.
If one considers the description given by the previous example, certen adjevtives of French take a gender and a number,
whereas others to not have any inflectional feature. This is for example the case with fixed sequences like
de bonne humeur, which have a syntactic behavior very close to that of adjectives.
Such sequences were thus integrated into the French dictionary as invariable adjectives without inflectional features.
The problem is that if one wants to refer exclusively to this type of adjectives in a disambiguation grammar,
the symbol `<A>` is not appropriate, since it will recognize all adjectives. To circumvent
this difficulty, it is possible to deny an inflectional attribute by writing the
character `@` right before one of the possible values for thsi attribute. Thus, the symbol `<A:@m@p>`
recognizes all the adjectives which have neither gender nor a number. Using this operator, it is now possible
to write grammars like those in figure 7.19, which imposes the agreemend in gender and number between
a name and an adjective which suits [2].
This grammar will preserve the correct analysis of sentences like: *Les personnes de bonne humeur m'insupportent*.
Is is however recommended to limit the use of the operator `@`, because it harms the legibility of the grammars.
It is preferable to distinguish the labels which accept various inflectional combinations by means of discriminating
subcategories defined in the `discr` part.

Optional Codes

Tje optional syntactic and semantic codes are declared in the `cat` part. They can be used in ELAG grammars
like the other codes. The difference is that these codes do not intervene to decide if a label must be rejected or not.

[2] This grammar is not completely correct, because it eliminates for example
the correct analysis of the sentence: *J'ai reçu des coups de fil de ma mère hallucinants.*

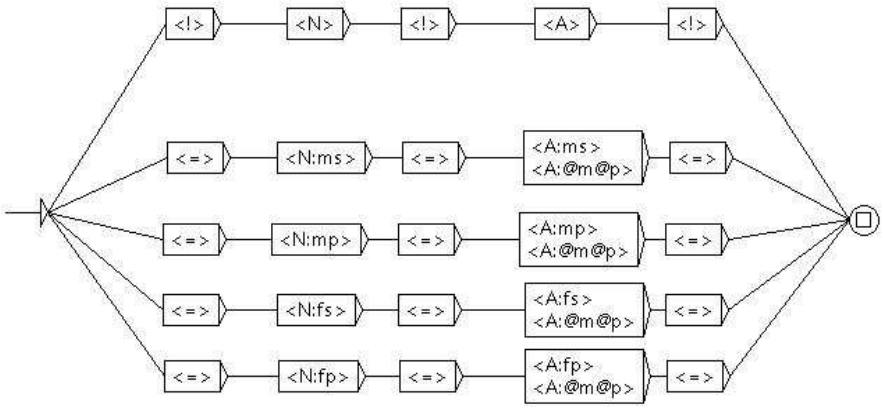

Figure 7.19: Grammaire ELAG vérifiant l'accord en genre et en nombre entre un nom et l'adjectif qui le suit

In fact optional codes are independent of the other codes, such as for example the attribute of the language level

(`z1`, `z2` or `z3`).

In the samme manner for inflectional codes, it is also possible to deny an inflectional attribute by writing the

`!` character right before the name of the attribute. Thus, with our example file, the `<A!gauche:f>`

recognizes all female adjectives which do not have the `g`[3].

All codes which are not declared in the `tagset.def` file are

ignored by ELAG. If a dictionary entry contains such a code, ELAG will produce a warning and will withdraw

the code for the entry.

Consequently, if two concurrent entries differ in the original text automaton only by not declared codes,

these entries will become indistinguable by the programs and will thus be unified in only one entry

in the resulting automaton.

Thus, the set of labels described in the file SSverb$tagset.def$ can be enough to reduce the ambiguity,

by factorizing words which differ only by not declared codes an this independently of the applied grammars.

For example, in the most complete version of the French dictionary, each individual use of a verb is characterized

[3]This code indicates that the adjective must
appear on the left of the nound to which it refers to, as it is the case for *bel*.

by a reference towards the lexicon grammar table which characterizes it . We have considered until now that these

informations are more relevant to syntax than to lexical analysis and we thus don't have them integrated into the

description of the sets of labels. They are thus automatically eliminated at the time when the text automaton

is loaded, which reduces the rate of ambiguities.

In order to distinguish the effects bound to the set of labels from those of the ELAG grammars, it is

advised to procees to a preliminiary stage of standardization of the text automaton before applying disambiguation

grammars to it.

This normalization is carried out by applying to the text autoaton a grammar not imposing any constraint,

like that of figure 7.20.

Note that this grammar is normally present in the Unitex distribution and precompiled in the file norm.rul.

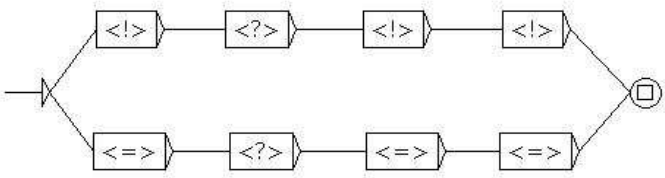

Figure 7.20: Grammaire ELAG n'exprimant aucune contrainte

The result of applying these grammars is that the original is cleaned of all the codes which are either not described

in the file `tagset.def`, or do not conforme to this description

(because of unknown grammatical categories r invalid combinations of inflectional features). By then replacing

the text automaton by this normalized automaton, one can be sure that later modifications of the automaton will be only due

to the effects of ELAG grammars.

7.3.7 Grammar Optimization

The compilation of ELAG grammars carried out by the `elagcomp` program

consists in building an automaton whose language is the whole of the sequences of lexical entries (or lexical interpretations

of a sentence) which are not refected by grammars. This task is complex and can take up much time. It is

however possible to appreciably accelerate it by observing certain principles at the time of writing gramars.

Limiting the number of branches in the *then* part

It is recommended to limit the number of *then* parts of a grammar to a minimum. This can reduce considerably the compile

time of a grammar. Generally, a grammar having many *then* parts can be rewritten with one or two *then* parts,

without a loss of legibility. It is for example the case of the grammar in figure 7.21, which imposes a

constraint between a verb and the pronoun which follows it.

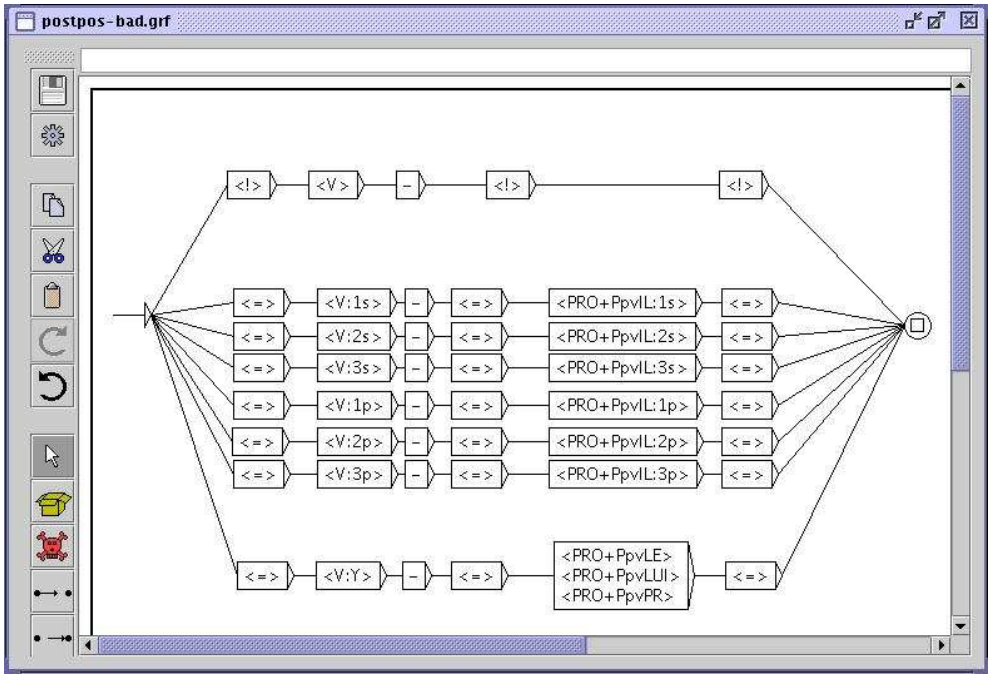

Figure 7.21: Grammaire ELAG vérifiant l'accord entre verbe et pronom

As one can see in figure 7.22, one can write an equivalent grammar by factorizing all the *then* party

into only one. The two grammars will have exactly the same effect on the text automaton, but the second one will

be compiled much more quickly.

Utilizing lexical symbols

It is beter to use lemmas only when it is abolutely necessary. That is particularly true for

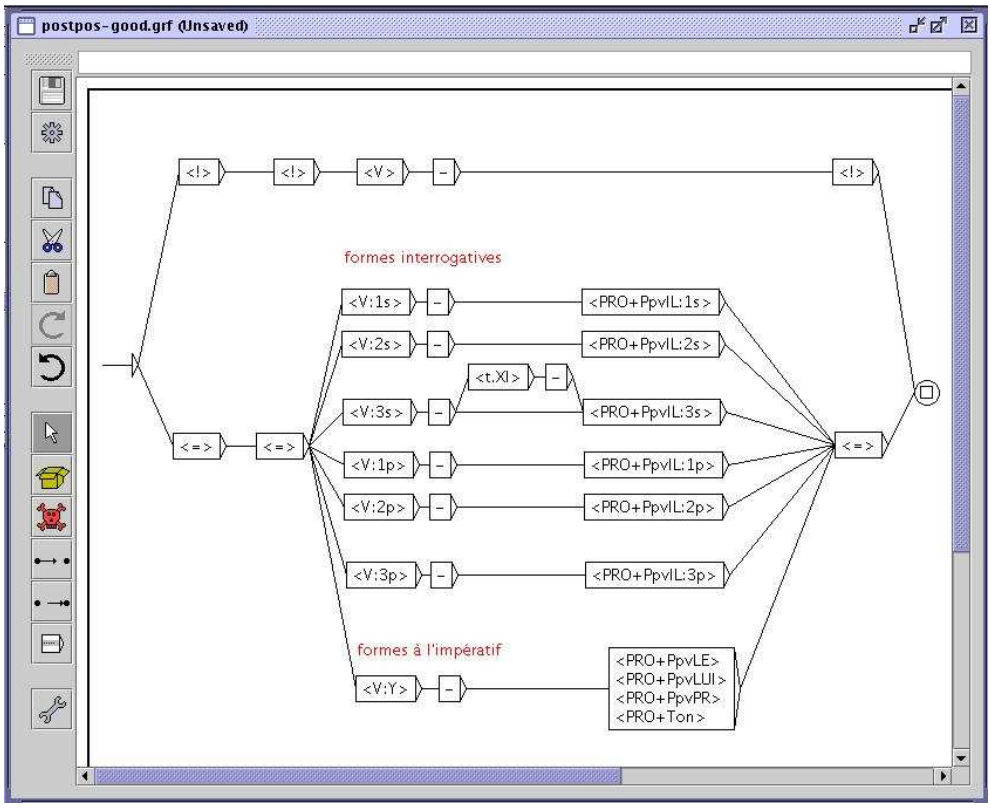

Figure 7.22: Grammaire ELAG optimisée vérifiant l'accord entre verbe et pronom

grammatical words, when their subcategories carry almost as much of information than the lemmas themselves.

If you despite everything use a lema in a symbol, it is recommended to specify its syntactic, semantic and inflectional

features as much as possible. For example, with the dictionaries provided for French, it is preferable to replace symbols

like `<je.PRO:1s>`, `<je.PRO+PpvIL:1s>` and

`<je.PRO>` with the symbol `<PRO+PpvI1:1s>`. Indeed, all these symbols are identical insofar as they can

recognize only the single entry of the dictionary `{je,PRO+PpvIL:1ms:1fs}`. However, as the program cannot deduce

this information automatically, if all these features are not specified, the program will consider nonexisting labels

such as `<je.PRO:3p>`, `<je.PRO+PronQ>` etc. in vain.

7.4 Manipulation of text automata

7.4.1 Displaying sentence automata

As we have seen above, the text automaton is in fact the collection of the sentence automata of a text. This structure can be represented using the format `.fst2`, used for representing the compiled grammars. .

This format does not allow to directly display the sentence automata. It is therefore necessary to use the program (`Fst2Grf`) for converting the sentence automaton into a graph that can be displayed. This program is called automatically when you select a sentence in order to generate the corresponding `.grf` file.

The generated `.grf` files are not interpreted in the same manner as the `.grf` files that represent the graphs that are constructed by the user. In fact, in a normal graph, the lines of a box are separated by the symbol `+`. In the graph of a sentence, each box is either a lexical unit without label or a dictionary entry enclosed by curly brackets. If the box only contains an unlabeled lexical unit, this appears alone in the box. If the box contains a dictionary entry, the inflected form is displayed, followed by the canonical form if it is different. The grammatical and inflectional information is displayed below the box as in the transductions.

Figure 7.23 shows the graph obtained for the first sentence of *Ivanhoe*. The words `Ivanhoe`, `Walter` and `Scott` are considered unknown words. The word `by` corresponds to two entries in the dictionary. The word `Sir` corresponds to two dictionary entries as well, but since the canonical form of these entries is `sir`, it is displayed because it differs from the inflected form by a lower case letter.

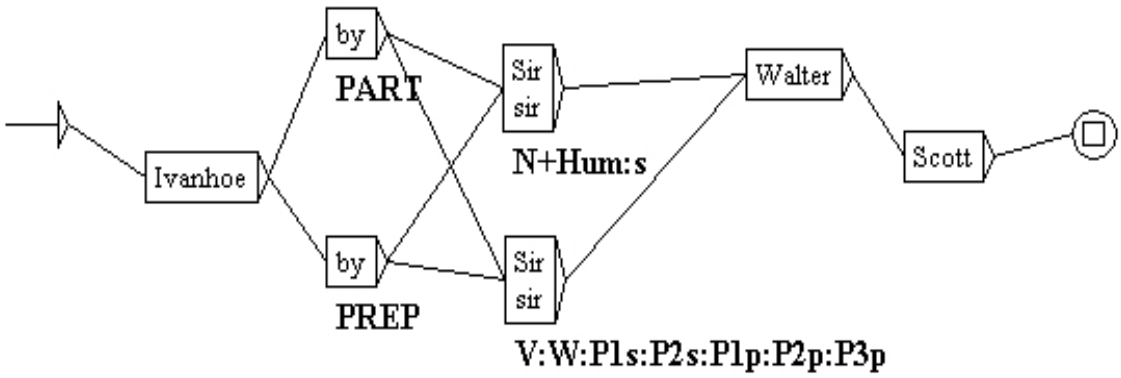

Figure 7.23: Automaton of the first sentence of *Ivanhoe*

7.4.2 Modify the text automaton

It is possible to manually modify the sentence automaton. You can add or erase boxes or transitions. When a graph is modified, it is saved to the text file `sentenceN.grf`, where N represents the number of the sentence.

When you select a sentence, if a modified graph exists for this sentence, this one is displayed. You can then reinitialize the automaton of that sentence by clicking on the botton "Reset Sentence Graph" (cf. figure 7.24).

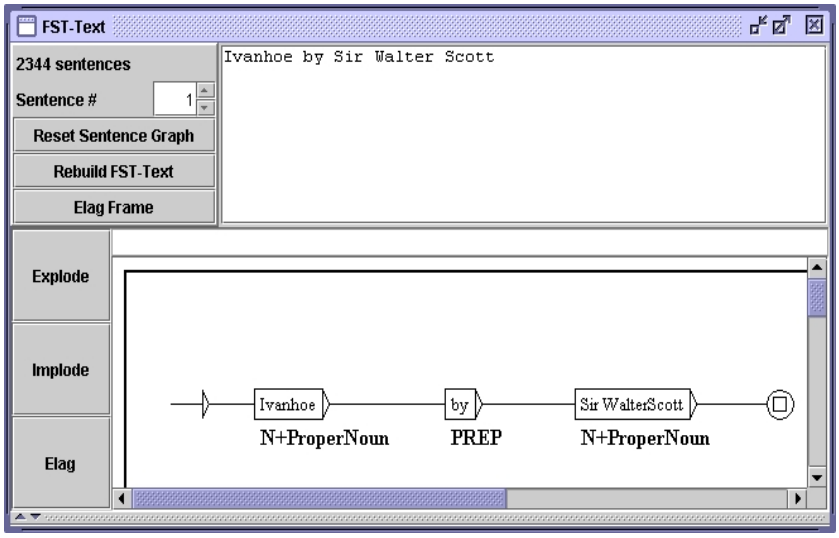

Figure 7.24: Modified sentence automaton

During the construction of the text automaton all the modified sentence graphs in the text file are erased.

NOTE: You can reconstruct the text automaton and keep your manual modifications. In order to do that, click on the button "Rebuild FST-Text". All sentences that have modifications are then replaced in the text automaton with their modified versions. The new text automaton is then automatically reloaded.

7.4.3 Parameters of presentation

The sentence automata are subject to the same presentation options as the graphs. They use the same colors and fonts as well as the antialiasing effect. In order to configure the appearance of the sentence automata, you modify the general configuration by clicking on "Preferences..." in the menu "Info". For further details, refer to the section 5.3.5.

You can also print a sentence automaton by clicking on "Print..." in the menu "FSGraph" or by pressing <Ctrl+P>. Make sure that the printer's page orientation is set to landscape mode. To configure this parameter, click on "Page Setup" in the menu "FSGraph".

Chapter 8

Lexicon Grammar

The tables of the lexicon grammar are a compact way of representing the syntactical properties of the elements of a language. Using the mechanism of template graphs, it is possible to automatically construct local grammars from these tables.

In the first part of the chapter the formalism of the tables is presented. The second part describes the template graphs and mechanism of automatically generating graphs starting from a lexicon grammar table.

8.1 The lexicon grammar tables

The lexicon grammar is a methodology developed by Maurice Gross based on the following principle: every verb has almost unique syntactical properties. Due to this fact, these properties need to be systematically described, since it is impossible to predict the exact behavior of a verb. These descriptions are represented by matrices where the rows correspond to the verbs and the columns to the syntactical properties. The considered properties are formal properties such as the number and nature of allowed complements of the verb and the different transformations the verb can undergo (passivization, nominalization, extraposition, etc.). The matrices, mostly called tables, are binary: a + sign at the intersection of a row and a column of a property if the verb has that property, a - sign if not.

This type of description has equally been applied to adjectives, predicative nouns, adverbs, as well as figurative expressions, all in multiple languages.

Figure 8.1 shows an example of a lexicon grammar table. The table concerns verbs that take a numerical complement.

8.2 Conversion of a table into graphs

8.2.1 Principle of template graphs

The conversion of a table into graphs is carried out by the mechanism of template graphs. The prinicple is the following: a graph that describes the possible constructions is con-

Table32NM.xls

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
1	N0 = Npc	N0 = N-hum	N0 = Nnr	N0 = V-n	Aux	32NM	N1 V	N1 = Nhum	N1 = N-hum	N1 = le fait Qu P	N1 = V-n	N1 = Dnum Nmes	Ppv = le	N-1 V N0	N0 V Adj	N0 V Dnum V-n	N0 V à N1	N-1 V N0 (<E>+à) N1	N1 = V-n	Exemple
2	-	+	-	-	avoir	accepter	-	-	+	-	-	-	+	-	-	-	-	-	-	Ce salon§accepte§vingt personnes
3	-	+	-	-	avoir	accueillir	-	-	+	-	-	-	+	-	-	-	-	-	-	Ce salon§accueille§vingt personnes
4	-	+	-	-	avoir	accuser	-	-	+	-	-	+	+	-	-	-	-	-	-	Max§accuse§80 kilos
5	-	+	-	-	avoir	accuser	-	-	+	+	-	-	-	-	-	-	-	-	-	Max§accuse§ses trente ans
6	-	+	-	-	avoir	admettre	-	-	+	-	-	-	+	-	-	-	-	-	-	On§admet§50 personnes dans cette salle
7	-	+	-	-	avoir	affecter	-	-	+	-	-	-	-	-	-	-	-	-	-	Ces cristaux§affectent§une forme géométrique
8	-	+	-	-	avoir	afficher	-	-	+	-	-	+	-	-	-	-	-	-	-	Les valeurs ont§affiché§un repli
9	-	+	-	-	avoir	aimer	-	-	+	+	-	-	+	-	-	-	-	-	-	La plante§aime§l'eau
10	-	+	-	-	avoir	approcher	+	-	+	-	-	+	+	-	-	-	-	-	-	Cette maison§approche§les deux millions
11	-	+	-	-	avoir	arpenter	-	-	-	-	-	+	+	-	-	+	+	+	-	Ce terrain§arpente§30 arpents
12	-	+	-	-	avoir	atteindre	-	-	+	-	-	+	+	-	-	-	-	-	-	Max§atteint§80 kilos
13	+	+	+	-	avoir	avoir	-	-	+	-	-	+	+	-	-	-	-	-	-	Max§a§(une soeur+une voiture+des sous)
14	-	+	-	-	avoir	avoisiner	-	-	+	-	-	+	+	-	-	-	-	-	-	Ce sac§avoisine§les 20 kg.
15	-	+	-	-	avoir	battre	+	-	+	-	-	-	+	-	-	-	-	-	-	La montre§bat§les secondes
16	-	+	-	-	avoir	cacher	-	-	+	-	-	-	-	-	-	-	-	-	-	Son calme§cache§(son+une grande)angoisse
17	-	+	-	-	avoir	caler	-	-	+	-	-	+	+	-	+	-	-	-	-	Ce bateau§cale§80 cm

c32NM

Figure 8.1: Lexicon Grammar Table 32NM 32NM

structed. This graph refers to the columns of the table in the form of variables. Afterwards, for each line of the table a copy of this graph is constructed where the variables are replaced with the contents of the cell at the intersection of the column and the line in question.

If a cell of the table contains the + sign, the corresponding variable is replaced by <E>. If the cell contains the - sign, the box containing the corresponding variable is removed, at the same time making the paths through that box unavailable. In all other cases the variable is replaced by the contents of the cell.

8.2.2 Format of the table

The lexicon grammar tables are usually represented with the aid of a spreadsheet like OpenOffice.org Calc. To make them usable with Unitex, the tables have to be encoded in Unicode text format in accordance with the following convention: the columns need to be separated by a tab and the lines by a newline. To convert a table with OpenOffice.org Calc, save it in text format (extension .csv). The program then allows to parameterize the saving of the file with a window like the one in figure XXX.

Select Unicode and tab as separator for columns and leave the field "text separator" empty.

During the generation of the graphs, Unitex skips the first line, considering it to be the headings of the columns. It is therefore necessary to ensure that the headings of the columns occupy exactly one line. If there is no line for the heading, the first line of a table should be ignored, and if there are multiple heading lines, from the second line on they will be

interpreted like lines of the table.

8.2.3 The template graphs

The template graphs are the graphs in which the variables appear that refer to the columns of a table of the lexicon grammar. This mechanism is usually used with syntactical graphs, but nothing prevents the construction of template graph for inflection, preprocessing, or for normalization.

The variables that refer to columns are formed with the @ symbol followed by the name of the column in capital letters (the columns are named starting with A).

Example:@C refers to the third column of the table

Whenever a variable needs to be replaced by a + or -, the - sign corresponds to the removal of a path through that variable. It is possible to carry out the inverse operation by putting an exclamation mark in front of the @ symbol. In that case, whenever the variable refers to the + sign, the path is removed. If the variable refers neither to the + sign nor the - sign, it is replaced by the contents of the cell.

There is also the special variable @% which is replaced by the number of the line in the table. The fact that its value is different for each line allows for its use as simple characterization of the line. That variable is not affected by an exclamation point to the left of it.

Figure 8.2 shows an example of a template graph designed to be applied to the lexicon grammar table 31H presented in figure 8.3.

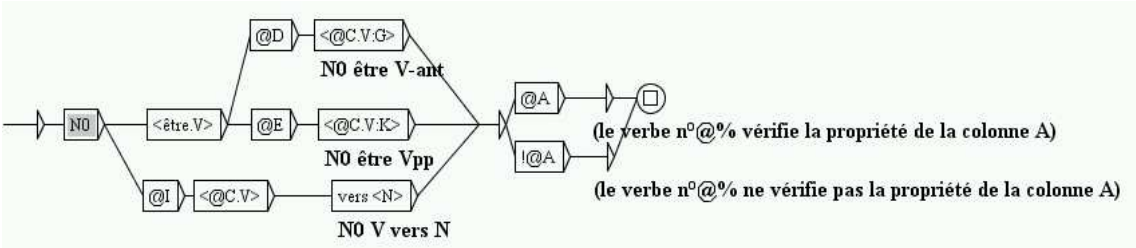

Figure 8.2: Example of a template graph

8.2.4 Automatic generation of graphs

In order to be able to generate graphs from a template graph and a table, first of all the table needs to be opened by clicking on "Open..." in the menu "Lexicon-Grammar" (see figure 8.4). The table needs to be in Unicode text format.

Table31H.xls

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	N0 = V-n	Aux	31H	N0 est V-ant	N0 est Vpp	N0pc lui V	N0 V de N0pc	Nhum V sur ce point	N0 V vers N	il V N0 W	idée Loc esprit	Nhum Loc Nabs	N0 = N-hum	N0 = V-n	Exemple
2	-	avoir	abandonner	-	-	-	-	-	-	-	-	-	-	-	Paul a§abandonné§
3	-	avoir	abuser	-	-	-	-	+	-	-	-	-	-	-	Max§abuse§
4	-	avoir	acquiescer	-	-	-	+	+	-	-	-	-	-	-	Max a§acquiescé§(E+de la tête)
5	-	avoir	adouber	-	-	-	-	-	-	-	-	-	-	-	Paul§adoube§|échecs|
6	-	avoir	agioter	-	-	-	-	-	-	+	-	-	-	-	Max§agiote§sur les changes
7	-	avoir	agoniser	+	-	-	-	-	-	+	-	-	+	-	Max§agonise§
8	-	avoir	archaïser	+	-	-	-	+	-	-	-	-	-	-	Cet auteur§archaïse§volontiers
9	-	avoir	arquer	-	-	-	+	-	+	-	-	-	-	-	Max a§arqué§toute la journée
10	-	être	arriver	-	+	-	-	-	-	+	-	+	-	-	Max est§arrivé§
11	-	avoir	atermoyer	-	-	-	-	+	-	+	-	+	-	-	Max§atermoie§
12	+	avoir	badauder	-	-	-	-	-	+	-	+	-	-	badaud	Max§badaude§

c31H

Figure 8.3: Lexicon grammar table 31H

Figure 8.4: Menu "Lexicon-Grammar"

The selected table is then opened in a window (see figure figure 8.5).

To automatically generate graphs from a template graph, click on "Compile to GRF..." in the menu "Lexicon-Grammar". The window in figure 8.6 shows this

In the field "Reference Graph (in GRF format)", enter the name of the template graph to be used. In the field "Resulting GRF grammar", enter the name of the main graph that will be generated. This main graph is a graph that refers to all graphs that are going to be generated. When launching a search in a text with that graph, all generated graphs are simultaneously applied.

The field "Named of produced subgraphs" is used to set the name of each graph that will be generated. It is a good idea to enter a name containing @%, because for each line of the table, @% will be replaced the line number, which guarantees that each graph name will be

E:\Table31H.txt

N0 =: V-n	Aux	31H	N0 est V-ant	N0 est Vpp	N0pc lui V	N0 V de N0..	Nhum V sur..	N0 V vers N	il V N0 W	i
-	avoir	abando...	-	-	-	-	-	-	-	
-	avoir	abuser	-	-	-	-	+	-	-	
-	avoir	acquie...	-	-	-	+	+	-	-	
-	avoir	adouber	-	-	-	-	-	-	-	
-	avoir	agioter	-	-	-	-	-	-	+	
-	avoir	agoniser	+	-	-	-	-	-	+	
-	avoir	archaïser	+	-	-	-	+	-	-	
-	avoir	arquer	-	-	-	+	-	+	-	
-	être	arriver	-	+	-	-	-	-	+	
-	avoir	atermoyer	-	-	-	-	+	-	+	
+	avoir	badauder	-	-	-	-	-	+	-	
-	avoir	baisser	-	-	-	-	+	-	-	
-	avoir	bambocher	-	-	-	-	-	-	+	
-	avoir	bander	-	-	-	+	-	-	+	
-	avoir	barouder	-	-	-	-	-	-	+	
-	avoir	batifoler	+	-	-	-	-	-	+	
-	avoir	bécher	-	-	-	-	+	-	-	
+	avoir	bêtifier	+	-	-	-	+	-	+	
+	avoir	bigler	-	-	+	+	-	+	+	
-	avoir	boiter	-	-	-	+	-	+	+	
-	avoir	boitiller	+	-	-	+	-	+	+	
+	avoir	bouffo...	-	-	-	-	-	-	+	

Figure 8.5: Displaying a table

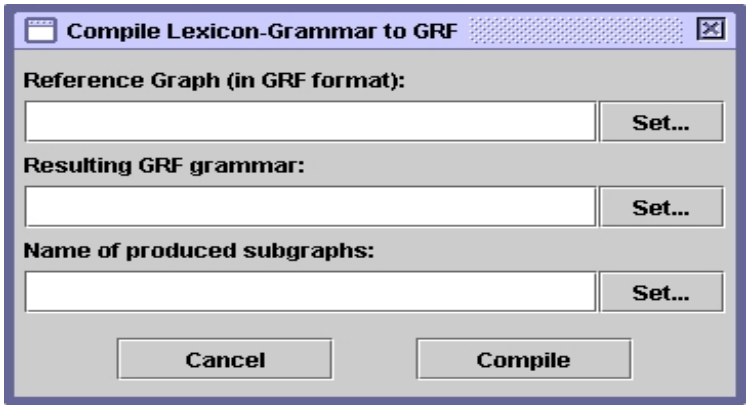

Figure 8.6: Configuration of the automatic generation of graphs

unique. For example, if the main graph is called "`TestGraph.grf`" and if subgraphs are called "`TestGraph_@%.grf`", the graph generated from the 16th line of the line of the table will be named "`TestGraph_0016.grf`".

Figures 8.7 and 8.8 show two graphs generated by applying the template graph of figure 8.2 to table 31H.

Figure 8.9 shows the resulting main graph.

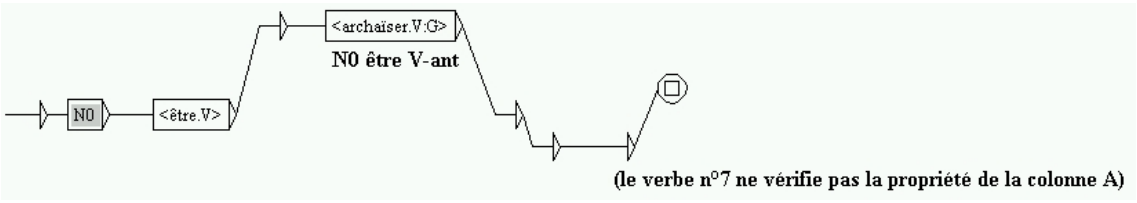

Figure 8.7: Graph generated for the verb `archaïser`

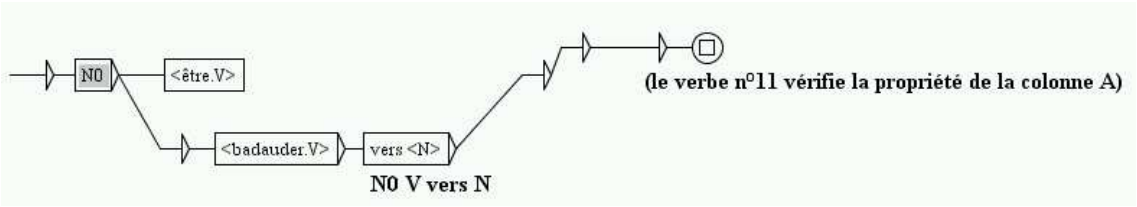

Figure 8.8: Graph generated for the verb `badauder`

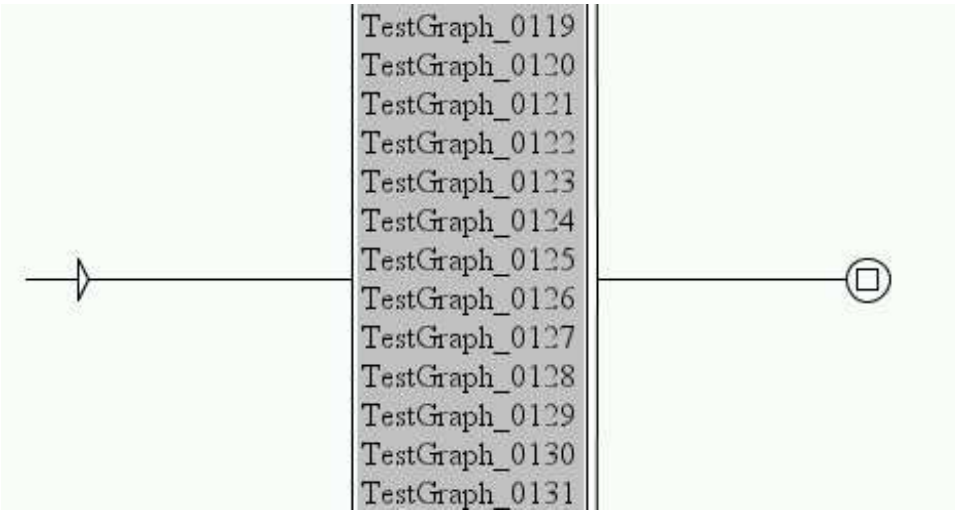

Figure 8.9: Main graph referring to all generated graphs

Chapter 9

Use of external programs

This chapter presents the use of the different programs of which Unitex is composed. These programs, which can be found in the directory `Unitex/App`, are automatically called by the interface. It is possible to see the commands that have been executed by clicking on the menu "Info" on the "Console". It is also possible to see the options of the different programs by selecting "Help on commands" in the menu "Info".

ATTENTION: a number of programs use the text directory (`my_text_snt`). This directory is created by the graphical interface after the normalization of the text. If you work with the command line, you have to create the directory manually before the execution of the program `Normalize`.

ATTENTION (2): whenever a parameter contains spaces, it needs to be enclosed in quotation marks so it will not be considered as multiple parameters.

9.1 CheckDic

```
CheckDic dictionary type
```

This program carries out the verification of the format of a dictionary of type DELAS or DELAF. The parameter `dictionnary` corresponds to the name of the dictionary that is to be verified. The parameter `type` can take the value `DELAS` or `DELAF` depending on the format of the dictionary to be verified.

The program checks the syntax of the lines of the dictionary. It also creates a list of all characters occuring in the inflected and canonical forms of words, the list of grammatical and syntactic codes , as well as the list of inflectional codes used. The results of the verification are stored in a file called `CHECK_DIC.TXT`.

9.2 Compress

```
Compress dictionary [-flip]
```

This program takes a DELAF dictionary as a parameter and compresses it. The compression of a dictionary `dico.dic` produces two files:

- `dico.bin`: a binary file containing the minimum automaton of the inflected forms of the dictionary
- `dico.inf`: a text file containing the compressed forms allowing the reconstruction of the dictionary lines from the inflected formes contained in the automaton.

For more details on the format of these files, see chapter 10. The optional parameter `-flip` indicates that the inflected and canonical forms should be inversed in the compressed dictionary. This option is used to construct an inverse dictionary which is necessary for the program `Reconstrucao`.

9.3 Concord

```
Concord index font fontsize left right order mode alph [-thai]
```

This program takes an index file of the concordance produced by the program `Locate` and produces a concordance. It is also possible to produce a modified text version taking into account the transductions associated with the occurrences. Here is the description of the parameters:

- `index`: name of the concordance file. It is necessary to specify the entire file path, since Unitex uses it to determine for which text the concordance is to be constructed.
- `font`: name of the typeface if the concordance is in HTML format. This value is ignored if the concordance is not in HTML format.
- `fontsize`: size of the typeface if the concordance is in HTML format. Like the parameter `font`, it is also ignored, if the concordance is not in HTML format.
- `left`: number of characters to the left of the occurrences. In Thai mode, this means the number of non-diacritic characters.
- `right`: number of characters (non-diacritic in Thai mode) to the right of the occurrences. If the occurrence is shorter than this value, the concordance line is displayed so that the right context is equal to `right`. If the occurrence has a length longer than the characters defined by `right`, it is nevertheless saved as whole.
- `order`: indicates the mode to be used for sorting the lines of the concordance. The possible values are:
  - `TO`: order in which the occurrences appear in the text;
  - `LC`: left context, occurrence;
  - `LR`: left context, right context;

  - `CL`: occurrence, left context;
  - `CR`: occurrence, right context;
  - `RL`: right context, left context;
  - `RC`: left context, occurrence.
  - `NULL`: does not specify any sorting mode. This option should be used if the text is to be modified instead of constructing a concordance.

  For details on the sorting modes, see section 4.8.2.

- `mode`: indicates in which format the concordance is to be produced. The four possible formats are:
  - `html`: produces a concordance in HTML format encoded in UTF-8;
  - `text`: produces a concordance in Unicode text format;
  - `glossanet`: produces a concordance for GlossaNet in HTML format. The HTML file is encoded in UTF-8;
  - `name_of_file`: indicates to the program that it is supposed to produce a modified version of the text and save it in a file named `name_of_file` (see section 6.6.3).
- `alph`: alphabet file used for sorting. The value NULL indicates the absence of an alphabet file.
- `-thai`: this parameter is optional. It indicates to the program that it is processing a Thai text. This option is necessary to ensure the proper functioning of the program in Thai.

The result of the application of this program is a file called `concord.txt` if the concordance was constructed in text mode, a file called `concord.html` if the mode was `html` or `glossanet`, and a text file with the name defined by the user of the program if the program has constructed a modified version of the text.

In `html` mode, the occurrence is coded as a link. The reference associated with this link is of the form `<a href="X Y Z">`. `X` and `Y` represent the beginning and ending positions of the occurrence in characters in the file `name_of_file.snt`. `Z` represents the number of the sentence in which the occurrence was found.

9.4 Convert

```
Convert src [dest] mode text_1 [text_2 text_3 ...]
```

This program allows to change the text file encoding. The `src` parameter indicates the input encoding. The optional `dest` parameter indicates the output encoding. By default,

the output encoding is `LITTLE-ENDIAN`. The possible values for these parameters are the following:

`FRENCH`
`ENGLISH`
`GREEK`
`THAI`
`CZECH`
`GERMAN`
`SPANISH`
`PORTUGUESE`
`ITALIAN`
`NORWEGIAN`
`LATIN` (default)
`windows-1252`: Microsoft Windows 1252 - Latin I code page (Western Europe & USA)
`windows-1250`: Windows 1250 code page - Central Europe
`windows-1257`: Microsoft Windows 1257 Code Page - Baltic Countries
`windows-1251`: Microsoft Windows 1251 code page - Cyrillic
`windows-1254`: Microsoft Windows 1254 code page - Turc
`windows-1258`: Microsoft Windows 1258 code page - Viet Nam
`iso-8859-1` : ISO 8859-1 code page - Latin 1 (Western Europe & USA)
`iso-8859-15` : ISO 8859-15 code page - Latin 9 (Western Europe & USA)
`iso-8859-2` : ISO 8859-2 code page - Latin 2 (Eastern and Central Europe)
`iso-8859-3` : ISO 859-3 code page - Latin 3 (Southern Europe)
`iso-8859-4` : ISO 859-4 code page - Latin 4 (Northern Europe)
`iso-8859-5` : ISO 8859-5 code page - Cyrillic
`iso-8859-7` : ISO 8859-7 code page - Greek
`iso-8859-9` : ISO 8859-9 code page - Latin 5 (Turkish)
`iso-8859-10` : ISO 8859-10 code page - Latin 6 (Nordic)
`next-step` : NextStep code page
`LITTLE-ENDIAN`
`BIG-ENDIAN`

NOTE: There is an additional mode for the `dest` parameter with the value `UTF-8`, which indicates to the program that it must convert the files from Unicode Little-Endian into UTF-8 files.

The parameter `mode` specifies how to manage the source and destination file names. The possible values are as follows:

`-r`: conversion deletes the source files
`-ps=PFX`: the source files are renamed with the prefix `pfx` (`toto.txt` → `pfxtoto.txt`)
`-pd=PFX`: the destination files destination are renamed with the prefix `PFX`
`-ss=SFX`: the source files are renamed with the suffix `SFX` (`toto.txt` → `totosfx.txt`)
`-sd=SFX`: the destination files are renamed with the suffix `SFX`

The parameters `text_i` are the names of the files to be converted.

9.5 Dico

```
Dico text alphabet dic_1 [dic_2 ...]
```

This program applies dictionaries to a text. The text has to be split up into lexical units by the program `Tokenize`. The dictionaries need to be compressed with the program `Compress`. `text` represents the complete file path, without omitting the extension `.snt`. `dic_i` represents the file path of a dictionary. The dictionary must have the extension `.bin`. It is possible to give priorities to the dictionaries. For details see section 3.6.1.

The program `Dico` produces the following four files, and saves them in the directory of the text:

- `dlf`: dictionary of simple words in the text;
- `dlc`: dictionary of compound words in the text;
- `err`: list of unknown words in the text;
- `stat_dic.n`: file containing the number of simple words, the number of compound words, and the number of unknown words in the text.

NOTE: the files `dlf`, `dlc` and `err` are not sorted. Use the program `SortTxt` to sort them.

9.6 Elag

```
Elag txtauto -l lang -g rules -o output [ -d dir]
```

This program takes a text automaton `txtauto` and applies disambiguation rules to it. The parameters are as follows:

- `txtauto`: the text automaton in `.fst2` format
- `lang`: the configuration file for ELAG for the language considered
- `rule`: the file of rules compiled in the `.rul` format
- `output`: the output text automaton
- `dir`: this optional parameter indicates the directory in which rules ELAG are to be found

9.7 ElagComp

```
ElagComp [ -r ruleslist | -g grammar ] -l lang [ -o output ] [ -d to rulesdir
```

This program compiles an ELAG grammar whose name is `grammar`, or all the grammars specified in the file `ruleslist`. The result is stored in a file `output` which could be used by the program `elag`.

- `ruleslist`: listing file of ELAG grammars
- `lang`: the ELAG configuration file for the chosen language
- `output`: (optional) name of the output file. By default, the output file is identical to `ruleslist`, except for the extension which is `.rul`
- `rulesdir`: this optional parameter indicates the directory in which ELAG rules are to be found

9.8 Evamb

```
Evamb [ -imp | -exp ] [ -o ] fstname [ -n sentenceno ]
```

This program calculates an average rate of ambiguity for the whole text automaton `fstname`, or right on the sentence specified by `sentenceno`. If the parameter `-imp` is specified, the program carries out calculation on a form the automaton known as *compacte* in which inflectional ambiguities are not taken into account. If the parameter `-exp` is specified, all inflectional ambiguities are considered; We then speak of the *developed form* of the text automaton. Thus, the entry `aimable, A:ms:f` will count only once with `-imp`, and twice with `-exp`. The text automaton is not modified by this program.

9.9 ExploseFst2

```
ExploseFst2 txtauto -o out
```

This program calculates , the *expanded form* of the text automaton `txtauto` and stores in `out`.

9.10 Extract

```
Extract yes/no text concordance result
```

This program takes a text and a concordance as parameters. If the first parameter is `yes`, the program extracts all sentences from the text that have at least one occurrence in the concordance. If the parameter is `no`, the program extracts all sentences that do not

contain any occurrences in the concordance. The parameter `text` represents the complete path of the text file, without omitting the extension .snt. The parameter `concordance` represents the complete path of the concordance file, without omitting the extension .ind. The parameter `result` represents the name of the file in which the extracted sentences are to be saved.

The result file is a text file that contains all extracted sentences, one sentence per line.

9.11 Flatten

```
Flatten fst2 type [depth]
```

This program takes any grammar as its parameter and tries to transform it into a finite state transducer. The parameter `fst2` specifies the grammar to transform. The parameter `type` specifies which kind of grammar the result grammar should be. If this parameter is `FST`, the grammar is "unfolded" to maximum depth and is truncated if there are further calls to sub-graphs. The result is a grammar in `.fst2` format that does only contain a single finite state transducer.

If the parameter is `RTN`, the calls to sub-graphs that could remain after the transformation are left as they are. The result is therefore a finite state transducer in the best case, and an optimized grammar strictly equivalent to the original grammar otherwise. The optional parameter `depth` specifies the maximum depth of nest of the sub-graphs that are generated by the program. The default value is 10.

9.12 Fst2Grf

```
Fst2Grf text_automaton sentence
```

This program extracts an automaton of a sentence in `.grf` format from the automaton of a text. The parameter `text_automaton` represents the complete path of the automaton file of the text from which a sentence is to be extracted. This file is called `text.fst2` and is stored in the directory of the text. The parameter `sentence` indicates the number of the sentence to extract.

The program produces the following two files and saves them in the directory of the text:

- `cursentence.grf`: graph representing the automaton of the sentence
- `cursentence.txt`: text file containing the sentence.

9.13 Fst2List

```
Fst2List [ -o out][-p s/f/d][-[a/t ] s/m][-f s/a][-s "L, [R]"][-s0 "Str"][-v]
[ -r[s/l/x]  "L, [R ]" ] [ -l line # ] [ -I subname ] * [ -c SS=0xxxx ] * fnam
```

This program takes a file `.fst2` and produces the list of the sequences recognized by this grammar. The parameters are as follows:

- `fname`: name of the grammar, with the extension `.fst2`;
- `-o out`: the name of the output file. By default, this file is named `lst.txt`;
- `-[a/t] s/m`: specifies if (`a`) or not (`t`) the outputs of the grammar are taken into account. `s` indicates that there is one initial state, while `m` indicates that there are several (this mode is useful in Korean). By default, this parameter is set to `-a s`;
- `-l line#`: maximum number of lines to be written in the output file;
- `-i subname`: indicates that the recursive exploration must stop when the graph `subname` is encountered. This parameter can be used several times, in order to specify several stop graphs;
- `-p s/f/d`: `s` lists the paths of each subgraph of the grammar; `f` (default) lists the paths of the main grammar; `d` lists the paths by adding the nesting depth of calls to subgraphs;
- `-c SS=0xXXXX`: replaces the symbol `SS` when it appears between parentheses by a Unicode character (`0xXXXX`);
- `-s "L[, R]"`: specifies the left (`l`) and right (`r`) delimiters which will surround the items. By default, these delimiters are empty;
- `-s0 "Str"`: if the outputs of the grammar are taken into account, this parameter specifies the sequence `Str` which will separate an input from its output. By default, there is no separator;
- `-f a/SS`: if outputs of the grammar are taken into account, this parameter specifies the format of the generated lines: `in0 in1 out0 out1` (`s`) or `in0 out0 in1 out1` (`a`). The default value is `s`;
- `-v`: this parameter produces the posting of information messages;
- `-r[s/l/x] "L,[R]"`: this parameter specifies how cycles should be presented. `L` and `R` specify delimiters. In the graph in figure 9.1, the results shown were obtained with the delimiter settings `L="[` and `R="]*"`:

```
-rs "[,]" il fait [très|CO]CO::très (CO specifies a label generated by the
-rs "[,]" il fait très [Loc0]Loc0très Loc0beau (Loc0 specifies a label gener
-rs "[,]" il fait [très très ]* il fait très beau
```

Figure 9.1: Graph with Cycle

9.14 Fst2Txt

```
Fst2Txt text fst2 alphabet mode [-char_by_char]
```

This program applies a transducer to a text at the preprocessing stage, when the text has not been split into lexical units yet. The parameters of the program are the following:

- `text`: the text file to modify, with the extension `.snt`;
- `fst2`: the transducer to apply;
- `alphabet`: the alphabet file of the language of the text;
- `mode`: the application mode of the transducer. The two possible modes are `-merge` and `-replace`;
- `-char_by_char`: this optional parameter permits the application of the transducer in "character by character" mode. This option is used for texts in Asian languages.

This program modifies the text file given as a parameter.

9.15 Grf2Fst2

```
Grf2Fst2 graph [y/n]
```

This program compiles a grammar into a file `.fst2` (for more details see section 6.2). The parameter `graph` denotes the complete path of the main graph of the grammar, without omitting the extension `.grf`. The second parameter is optional. It indicates to the program whether the grammar needs to be checked for errors or not. Per default, the program carries out this error check.

The result is a file that carries the same name as the graph passed to the program as a parameter, but with the extension `.fst2`. This file is saved in the same folder as `graph`.

9.16 ImploseFst2

```
textauto -o out
```

This program calculates the *compact* form of the text automaton and stores it in `out`.

9.17 Inflect

```
Inflect delas result dir
```

This program carries out the automatic inflection of a DELAS dictionary. The parameter `delas` indicates the name of the dictionary to inflect. The parameter `result` indicates the name of the dictionary to be generated. The parameter `dir` indicates the complete file path of the directory in which the inflection transducers are that the `delas` dictionary refers to.

The result of the inflection is a DELAF dictionary saved under the name indicated by the parameter `result`.

9.18 Locate

```
Locate text fst2 alphabet s/l/a i/m/r n [-thai] [-space]
```

This program applies a grammar to a text and costructs an index file of the found occurrences. The following are its parameters:

- `text`: complete path of the text file, without omitting the extension `.snt`;
- `fst2`: complete path of the grammar, without omitting the extension `.fst2`;
- `alphabet`: complete path of the alphabet file;
- `s/l/a`: parameter indicating whether the search should be carried out in mode *shortest matches* (`s`), *longest matches* (`l`) or *all matches* (`a`);
- `i/m/r`: parameter indicating the application mode of the transductions: mode MERGE (`m`) or mode REPLACE (`r`). `i` indicates that the program should not take into account transductions;
- `n`: parameter indicating how many occurences to search for; The value `all` indicates that all occurrences need to be extracted;
- `-thai`: optional parameter necessary for searching a Thai text;
- `-space`: optional parameter indicating that the search should be performed beyond spaces. This parameter should only be used to carry out morphological searches.

This program saves the references to the found occurrences in a file called `concord.ind`. The number of occurrences, the number of units coverered by those occurrences, as well as the percentage of recognized units within the text are saved in a file called `concord.n`. These two files are stored in the directory of the text.

9.19 MergeTextAutomaton

```
MergeTextAutomaton automaton
```

This program reconstructs the text automaton `automaton` taking into account the modification manually conducted. In addition to that, if the program finds a file `sentenceN.grf` in the same directory as `automaton`, it replaces the automaton of sentence `N` with the one represented by `sentenceN.grf`. The file `automaton` is replaced by the new text automaton. The old text automaton is backed up in a file called `text.fst2.bck`.

9.20 Normalize

```
Normalize txt
```

This program carries out a normalization of text separators. The separators are space, tab, and newline. Every sequence of separators that contains at least one newline is replaced by a unique newline. All other sequences of separators are replaced by a single space.

This program also verifies the syntax of the lexical tables present in the text. All sequences in curly brackets are either the sentence delimiter `{S}`, or a valid line of DELAF (`{aujourd'hui,.ADV}`). If the program finds curly brackets that are used differently, it gives a warning and replaces them by square brackets (`[` and `]`). The parameter 'txt' represents the complete path of the text file. The program creates a modified version of the text that is saved in a file with the extension `.snt`.

9.21 PolyLex

```
PolyLex lang alph dic list out [info]
```

This program takes a file with unknown words `list` and tries to analyze each of the words as a compound obtained by combining simple words. The words that have at least one analysis are removed from the file of unknown words and the dictionary lines that correspond to the analysis are appended to the file `out`. The parameter `lang` determines the language to use. The two possible values are `GERMAN` and `NORWEGIAN`. The parameter `alph` represents the alphabet file to use. The parameter `dic` specifies which dictinary to consult for the analysis. The parameter `out` specifies the file to which the produced dictionary lines are to be written; if that file already exists, the produced lines are appended at the end of the file. The optional parameter `info` specifies a text file in which the information about the analysis will be written.

9.22 Reconstrucao

```
Reconstrucao alph concord dic reverse_dic pro res
```

This program generates a normalization grammar designed to be applied before the construction of an automaton for a Portugese text. The parameter `alph` specifies the alphabet file to use. The file `concord` represents a concordance which has to been produced by the application in MERGE mode to the considered text of a grammar that extracts all forms to normalize. This grammar is called `V-Pro-Suf`, and is stored in the directory `/Portuguese/Graphs/Normalization`. The parameter `dic` specifies which dictionary to use to find the canonical forms that are associated with the roots of the verbs. `reverse_dic` specifies the inverse dictionary to use to find the forms in future and conditional starting from canonical forms. These two dictionaries have to be in `.bin` format, and `reverse_dic` has to be obtained by compressing the dictionary of verbs in future and conditional with the parameter `-flip` (see section 9.2). The parameter `pro` specifies the grammar of reentry of the pronoms to use. `res` specifies the file `.grf` into which the normalization rules are to written.

9.23 Reg2Grf

```
Reg2Grf file
```

This program constructs a file `.grf` corresponding to the regular expression in file `file`. The parameter `file` represents the complete path to the file containing the regular expression. This file needs to be a Unicode text file. The program takes into account all characters up to the first newline. The result file is called `regexp.grf` and is saved in the same directory as `fic`.

9.24 SortTxt

```
SortTxt text [OPTIONS]
```

This program carries out a lexicographical sorting of the lines of the file `text`. `text` represents the complete path of the file to sort. The possible options are:

- `-y`: delete duplicates;
- `-n`: keep doubles;
- `-r`: sort in descending order;
- `-o file`: sort using the alphabet of the order defined by the file `file`. If this parameter is missing, the sorting is done according to the order of the Unicode characters;
- `-l file`: save the number of lines of the result file in the file `file`;
- `-thai`: option for sorting a Thai text.

The sort operation modifies the file `text`. By default, the sorting is performed in the order of the Unicode characters, removing duplicates.

9.25 Table2Grf

```
Table2Grf table grf result.grf [pattern]
```

This program automatically generates graphs from a lexicon-grammar `table` and the template graph `grf`. The name of the produced main graph of the grammar is `result.grf`.

If the parameter pattern if specified, all the produced subgraphs will be named according this pattern. In order to have unambiguous names, we recommend to include `@%` in the parameter (remember that `@%` will be replaced by the line number of the entry in the table). For instance, if you set the pattern parameter to `'subgraph-@%.grf'`, subgraphs names will be in the form `'subgraph-0013.grf'`. If the parameter pattern is not specified, subgraphs name are of the form `'result_i.grf'`, where `'result.grf'` specifies the result main graph.

9.26 TextAutomaton2Mft

```
TextAutomaton2Mft text.fst2
```

This program takes a text automaton `text.fst2` as a parameter and constructs the equivalent in the `.mft` format of Intex. The produced file is called `text.mft` and is encoded in Unicode.

9.27 Tokenize

```
Tokenize text alphabet [-char_by_char]
```

This program cuts the text into lexical units. The parameter `text` represents the complete path of the text file, without omitting the extension `.snt`. The parameter `alphabet` represents the complete path of the alphabet definition file of the language of the text. The optional parameter `-char_by_char` indicates whether the program is applied character by character, with the exception of the sentence separator `{S}` which is considered to be a single unit. Without this parameter set, the program considers a unit to be either a sequence of letters (the letters are defined by the file `alphabet`), or a character which is not a letter, or the sentence separator `{S}`, or a lexical label (`{aujourd'hui,.ADV}`).

The program codes each unit as a whole. The list of units is saved in a text file called `tokens.txt`. The sequence of codes representing the units now allows the coding of the text. This sequence is saved in a binary file named `text.cod`. The program also produces the following four files:

- `tok_by_freq.txt`: text file containing the units ordered by frequency;
- `tok_by_alph.txt`: text file containing the units ordered alphabetically;
- `stats.n`: text file containing information on the number of sentence separators, the number of units, the number of simple words and the number of numbers;

- `enter.pos`: binary file containing the list of newline positions in the text. The coded representation of the text does not contain newlines, but spaces. Since a newline counts for two characters and the space for a single one, it is necessary to know where there are newlines in the text if the posistions of the calcualted occurrences by the program `Locate` are to be synchronized with the text file. For this the file `enter.pos` is used by the program `Concord`. Thanks to this, when clicking on an occurrence in a concordance, it is correctly selected in the text.

All produced files are saved in the directory of the text.

9.28 Txt2Fst2

```
Txt2Fst2 text alphabet [-clean] [norm]
```

This program constructs an automaton of a text. The parameter `text` represents the complete path of a text file without omitting the extension `.snt`. The parameter `alphabet` represents the complete path of the alphabet file of the language of the text. The optional parameter `-clean` indicates whether the principle of conservation of the best paths (see section 7.2.4) should be applied. If the parameter `norm` is specified, it is interpreted as the name of a normalization grammar that is to be applied to the text automaton.

If the text is split into sentences, the program constructs an automaton for each sentence. If this is not the case, the program arbitrarily cuts the text into sequences of 2000 lexical units and produces an automaton for each of these sequences.

The result is a file called `text.fst2` which is saved in the directory of the text.

Chapter 10

File formats

This chapter presents the formats of files read or generated by Unitex. The formats of the DELAS and DELAF dictionaries have already been presented in sections 3.1.1 and 3.1.2.

NOTE: in this chapter the symbol ¶ represents the newline symbol. Unless otherwise indicated, all text files described in this chapter are encoded in Unicode Little-Endian.

10.1 Unicode Little-Endian encoding

All text files processed by Unitex have to be encoded in Unicode Little-Endian. This encoding allows the representation of 65536 characters by coding each of them in 2 bytes. In Little-Endian, the bytes are in lo-byte hi-byte order. If this order is reversed, we speak of Big-Endian. A text file encoded in Unicode Little-Endian starts with the special character with the hexadecimal value FEFF. The newline symbols have to be encoded by the two characters 000D and 000A.

Consider the following text:

```
Unitex¶
β-version¶
```

Here its representation in Unicode Little-Endian:

en-tête	U	n	i	t	e	x	¶	β
FFFE	5500	6E00	6900	7400	6500	7800	0D000A00	B203
-	v	e	r	s	i	o	n	¶
2D00	7600	6500	7200	7300	6900	6F00	6E00	0D000A00

Table 10.1: Hexadecimal representation of a Unicode text

The hi-bytes and lo-bytes have been reversed, which explains why the start character is encoded as FFFE in stead of FEFF, and 000D and 000A are 0D00 and 0A00 respectively.

10.2 Alphabet files

There are two kinds of alphabet files: a file which defines the characters of a language, and a file that indicates the sorting preferences. The first is called *alphabet*, the second *sorted alphabet*.

10.2.1 Alphabet

The alphabet file is a text file that describes all characters of a language, as well as the correspondences between capitalized and non-capitalized letters. This file is called `Alphabet.txt` and is found in the root of the directory of a language. Its presence is obligatory for Unitex to function.

Example: the English alphabet file has to be in the directory `.../English/`

Each line of the alphabet file must have one of the following three forms, followed by a newline symbol:

- #가힣 : a hash symbol followed by two characters X and Y which indicate that all characters between X and Y are letters. All these characters are considered to be in non-capitalized and capitalized form at the same time. This method is used to define the alphabets of Asian languages like Korean, Chinese or Japanese where there is no destinction between upper- and lower-case, and where the number of characters makes a complete enumeration very tedious;
- Ëë : two characters X and Y indicate that X and Y are letters and that X is equivalent in its capitalized and non-capitalized form.
- ᪨: a unique character X defines X as a letter in capitalized and non-capitalized form. This form is used to define an Asian punctuation mark.

For certain languages like French, it is possible that a lower case letter corresponds to multiple upper case letters, like for example é, which can have the upper case form E or É. To express this, it suffices to use multiple lines. The inverse is equally true: a capitalized letter can correspond to multiple lower case letters. Thus, the E can be the capitalization of e, é, è, ê or ë. Here an excerpt of the French alphabet file which defines the different letters e:

```
Ee¶
Eé¶
Éé¶
Eè¶
Èè¶
Eê¶
Êê¶
Eë¶
Ëë¶
```

10.2.2 Sorted alphabet

The sorted alphabet text file defines the sorting priorities of the letters of a language with which to sort with the program `SortTxt`. Each line of that file defines a group of letters. If a group of letters A is defined before a group of letters B, every letter of group A is of lower priority than every letter in group B.

The letters of a group are only distinguished if necessary. For example if the group of letters `eéèêë` has been defined, the word `ébahi` should be considered 'smaller' than `estuaire`, and also 'smaller' than `été`. Since the letters that follow `e` and `é` allow a classification of the words, it is not necessary to compare the letters `e` and `é` since they are of the same group. On the other hand, if the words `chantés` and `chantes` are to be sorted, `chantes` should be considered as 'smaller'. It is therefore necessary to compare the letters `e` and `é` to distinguish these words. Since the letter `e` appears first in the group `eéèêë`, it is considered to be 'smaller' than `chantés`. The word `chantes` should therefore be considered to be 'smaller' than the word `chantés`.

The sorted alphabet file allows the definition of equivalent characters. It is therefore possible to ignore the different accents as well as capitalization. For example, if the letters `b`, `c`, and `d` are to be ordered without considering capitalization and the cedilla, it is possible to write the following lines:

```
Bb¶
CcÇç¶
Dd¶
```

This file is optional. If no sorted alphabet file is specified, the program `SortTxt` creates a sorting in the order of the Unicode encoding.

10.3 Graphs

This section presents the two graph formats: the graphic format `.grf` and the compiled format `.fst2`.

10.3.1 Format .grf

A `.grf` file is a text file that contains presentation information in addition to information representing the contents of the boxes and the transitions of the graph. A `.grf` file begins with the following lines:

```
#Unigraph¶
SIZE 1313 950¶
FONT Times New Roman:  12¶
OFONT Times New Roman:B 12¶
BCOLOR 16777215¶
FCOLOR 0¶
ACOLOR 12632256¶
```

```
SCOLOR 16711680¶
CCOLOR 255¶
DBOXES y¶
DFRAME y¶
DDATE y¶
DFILE y¶
DDIR y¶
DRIG n¶
DRST n¶
FITS 100¶
PORIENT L¶
#¶
```

The first line `#Unigraph` is a comment line. The following lines define the parameter values of the graph presentation:

- `SIZE x y` : defines the width `x` and the height `y` of a graph in pixels;
- `FONT name:xyz` : defines the font used for displaying the contents of the boxes. `name` represents the name of the mode. `x` indicates if the text should be in bold face or not. If `x` is `B`, it indicates that it should be bold. For non-bold face, `x` should be a space. In the same way, `y` has the value `I` if the text should be italic, a space if not. `z` represents the size of the text;
- `OFONT name:xyz` : defines the mode used for displaying the transductions. The parameters `name`, `x`, `y`, and `z` are defined in the same way as `FONT`;
- `BCOLOR x` : defines the background color of the graph. 'x' represents the color in RGB format;;
- `FCOLOR x` : defines the desing color of the graph. 'x' represents the color in RGB format;
- `ACOLOR x` : defines the color used for drawing the lines of the boxes that correspond to the calls of sub-graphs. `x` represents the color in RGB format;
- `SCOLOR x` : defines the color used for writing in the comment box (the boxes that are not linked to any others). `x` represents the color in RGB format;
- `CCOLOR x` : defines the color used for drawing the selected boxes. `x` represents the color in RGB format;
- `DBOXES x` : this line is ignored by Unitex. It is conserved to ensure the compatibility with Intex graphs;
- `DFRAME x` : draws a frame around the graph if `x` is `y`, not if it is `n`;
- `DDATE x` : puts the date at the bottom of the graph if `x` is `y`, not if it is `n`;

- `DFILE x` : puts the name of the file at the bottom of the graph depending on whether `x` is `y` or `n`; em `DDIR x` : puts the complete path of the file at the bottom of the graph depending on whether `x` is `y` or `n`. This option takes effect only if the parameter `DFILE` has the value `y`;
- `DRIG x` : draws the graph from right to left or left to right depending on whether `x` is `y` or `n`;
- `DRST x` : this line is ignored by Unitex. It is conserved to ensure the compatibility with Intex graphs;
- `FITS x` : this line is ignored by Unitex. It is conserved to ensure the compatibility with Intex graphs;
- `PORIENT x` : this line is ignored by Unitex. It is conserved to ensure the compatibility with Intex graphs;
- `#` : this line is ignored by Unitex. It serves to specify the end of the header information.

The following lines give the contents and the position of the boxes in the graph. The following lines correspond to a graph recognizing a number:

```
3¶
"<E>" 84 248 1 2 ¶
"" 272 248 0 ¶
s"1+2+3+4+5+6+7+8+9+0" 172 248 1 1 ¶
```

The first line indicates the number of boxes in the graph, immediately followed by a newline. This number can not be lower than 2, since a graph always has an initial and a final state.

The following lines define the boxes of the graph. The boxes are numbered starting at 0. By convention, state 0 is the initial state and state 1 is the final state. The contents of the final state is always empty.

Each box in the graph is defined by a line that has the following format:

contents X Y N transitions ¶

contents is a sequence of characters enclosed in quotation marks that represents the contents of the box. This sequence can sometimes be preceded by an `s` if the graph is imported from Intex; this character is then ignored by Unitex. The contents of the sequence is the text that has been entered in the editing line for graphs. The following table shows the encoding of two special sequences that are not encoded in the same way as they are entered into the files `.grf`:

Sequence in the graph editor	Sequence in the file `.grf`
"	\"
\"	\\\"

Table 10.2: Encoding of special sequences

NOTE: The characters between < and > or between { and } are not being interpreted as a line separator. Thus the character + in the sequence `"le <A+Conc>"` is not interpreted like a line separator, since the pattern `<A+Conc>` is interpreted with priority.

X and *Y* represent the coordinates of the box in pixels. Figure 10.1 shows how these coordinates are interpreted by Unitex.

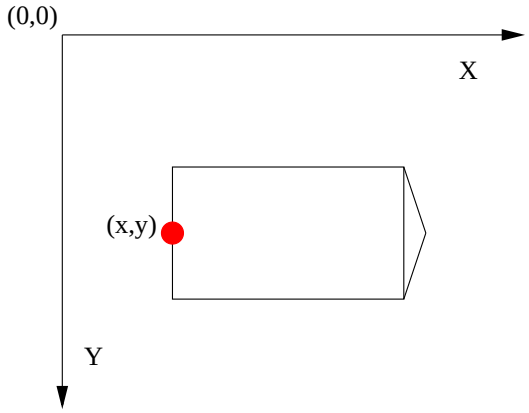

Figure 10.1: Interpretation of the coordinates of boxes

N represents the number of transitions that leave the box. This number is always 0 for the final state.

The transitions are defined by the numbers of boxes at which they point.

Every line of the box definition ends with a newline.

10.3.2 Format .fst2

An `.fst2` file is a text file that describes a set of graphs. Here an example of an `.fst2` file:

```
0000000002¶
-1 NP¶
: 1 1 ¶
: 2 2 -2 2 ¶
```

```
: 3 3 ¶
t ¶
f ¶
-2 Adj¶
: 6 1 5 1 4 1 ¶
t ¶
f ¶
%<E>¶
%the/DET¶
%<A>/ADJ¶
%<N>¶
%nice¶
@pretty¶
%small¶
f¶
```

The first line represents the number of graphs that are encoded in the file. The beginning of each graph is identified by a line that indicates the number and the name of the graph (`-1 NP` and `-2 Adj` in the file above).

The following lines describe the states of the graph. If the state is final, the line starts with the character `t` and with the character `:` if not. For each state, the list of transitions is a possibly empty sequence of entity pairs:

- the first entity indicates the number of the label or the sub-graph corresponding to the transition. The labels are numbered starting at 0. The sub-graphs are represented by negative entities, which explains why the numbers preceeding the names of the graphs are negativ;
- the second entity represents the number of the result state after the transition. In each graph, the states are numbered starting at 0. By convention the state 0 of a graph is its initial state.

Each definition line of a state terminates with a space. The end of each graph is marked by a line containing an `f` followed by a space.

The labels are defined after the last graph. If the line begins with the `@` character, the contents of the label is to be searched regardless of case. This information is only useful if the label is a word. If the line starts with a `%`, case variants are taken into account. If a label carries a transduction, the input and output sequences are separated by the `/` character (example: `the/DET`). By convention, the first label is always the empty word (`<E>`), even if that label is never used for any transition.

The end of the file is indicated by a line containing the character `f` followed by a newline.

10.4 Texts

This section presents the different files used to represent texts.

10.4.1 .txt files

The `.txt` files are text files encoded in Unicode Little-Endian. These files should not contain any opening or closing braces, except for those used to mark a sentence separator (`{S}`) or a valid lexical label (`{aujourd'hui,.ADV}`). The newline needs to be encoded with the two special characters with the hexadecimal values `000D` and `000A`.

10.4.2 .snt Files

The `.snt` files are `.txt` files that have been processed by Unitex. These files should not contain any tabs. They should also not contain multiple consecutive spaces or newlines. The only allowed braces in the `.snt` files are those of the sentence separator `{S}` and those of lexical labels (`{aujourd'hui,.ADV}`).

10.4.3 File text.cod

The file `text.cod` is a binary file containing a sequence of entities that represent the text. Each entity `i` reflects the token with index `i` in the file `tokens.txt`. These entities are encoded in four bytes.

NOTE: The tokens are numbered starting at 0.

10.4.4 The file tokens.txt

The file `tokens.txt` is a text file that contains the list of all lexical units of the text. The first line of this file indicates the number of units found in the file. The units are separated by a newline. Whenever a sequence is found in the text with capitalization variants, each variant is encoded as a distinct unit.

NOTE: the newlines that might be in the file `.snt` are encoded like spaces. Therefore there is never a unit encoding the newline.

10.4.5 The files tok_by_alph.txt and tok_by_freq.txt

These two files are text files that contain the list of lexical units sorted alphabetically or by frequency.

In the `tok_by_alph.txt file`, each line is composed of a unit, followed by a tab and the number of occurrences of the unit within the text.

The lines of the `tok_by_freq.txt` file are formed after the same principle, but the number of occurrences occurs after the tab and the unit.

10.4.6 The file enter.pos

This file is a binary file containing the list of positions of the newline symbol in the file `.snt`. Each position is the index in the file `text.cod` where a newline has been replaced by a space. These positions are entities that are encoded in 4 bytes.

10.5 Text Automaton

10.5.1 The file text.fst2

The file `text.fst2` is a special `.fst2` file that represents the text automaton. In that file, each sub-graph represents a sentence automaton. The areas reserved for the names of the sub-graphs are used to store the sentences from which the sentence automata have been constructed.

With the exception of the first label which is always the empty word (`<E>`), the lables have to be either lexical units or entries from DELAF in braces.

Example: Here the file that corresponds to the text *He is drinking orange juice.*

```
0000000001¶
-1 He is drinking orange juice. ¶
: 1 1 2 1 ¶
: 3 2 4 2 ¶
: 5 3 6 3 7 3 ¶
: 8 4 9 4 10 4 11 5 ¶
: 12 5 13 5 ¶
: 14 6 ¶
t ¶
f ¶
%<E>¶
%{He,he.N:s:p}¶
%{He,he.PRO+Nomin:3ms}¶
%{is,be.V:P3s}¶
%{is,i.N:p}¶
%{drinking,drinking.A}¶
%{drinking,drinking.N:s}¶
%{drinking,drink.V:G}¶
%{orange,orange.A}¶
%{orange,orange.N+Conc:s}¶
%{orange,orange.N:s}¶
%{orange juice,orange juice.N+XN+z1:s}¶
%{juice,juice.N+Conc:s}¶
%{juice,juice.V:W:P1s:P2s:P1p:P2p:P3p}¶
%.¶
f¶
```

10.5.2 The file cursentence.grf

The file `cursentence.grf` is generated by Unitex during the display of a sentence automaton. The program `FST2Grf` constructs a file `.grf` from the file `text.fst2` that represents a sentence automaton.

10.5.3 The file sentenceN.grf

Whenever the user modifies a sentence automaton, that automaton is saved under the name `sentenceN.grf`, where `N` represents the number of the sentence.

10.5.4 The file cursentence.txt

During the extraction of the sentence automaton, the text of the sentence is saved in the file called `cursentence.txt`. That file is used by Unitex to display the text of the sentence under the automaton. That file contains the text of the sentence, followed by a newline.

10.6 Concordances

10.6.1 The file concord.ind

The file `concord.ind` is the index of the occurrences found by the program `Locate` during the application of a grammar. It is a text file that contains the starting and end position of each occurrence, possibly accompanied by a sequence of letters if the concordance has been obtained by taking into account the possible transductions of the grammar. Here an example of a file:

```
#M¶
59 63 the[ADJ= greater] part¶
67 71 the beautiful hills¶
87 91 the pleasant town¶
123 127 the noble seats¶
157 161 the fabulous Dragon¶
189 193 the Civil Wars¶
455 459 the feeble interference¶
463 467 the English Council¶
568 572 the national convulsions¶
592 596 the inferior gentry¶
628 632 the English constitution¶
698 702 the petty kings¶
815 819 the certain hazard¶
898 902 the great Barons¶
940 944 the very edge¶
```

The first line indicates in which transduction mode the concordance has been constructed. The three possible values are:

- `#I` : the transductions have been ignored;
- `#M` : the transductions have been inserted into the recognize sequences (MERGE mode);
- `#R` : the transductions have replaced the recognized sequences (REPLACE mode)).

Each occurrence is described in one line. The lines start with the start and end position of the occurrence. These positions are given in lexical units.

If the file has the heading line `#I`, the end position of each occurrence is immediately followed by a newline. Otherwise, it is followed by a space and a sequence of characters. In REPLACE mode, that sequence corresponds to the transduction produced for the recognized sequence. In MERGE mode, it represents the recognized sequences into which the transductions have been inserted. In MERGE or REPLACE mode, this sequence is displayed in the concordance. If the transductions have been ignored, the contents of the occurrence is extracted from the text file.

10.6.2 The file concord.txt

The file `concord.txt` is a text file that represents a concordance. Each occurrence is encoded in a line that is composed of three character sequences separated by a tab, representing the left context, the occurrence (possibly modified by transductions) and the right context.

10.6.3 The file concord.html

The `concord.html` file is an `HTML` file that represents a concordance. This file is encoded in UTF-8.

The title of the page is the number of occurrences it contains. The lines of the concordance are encoded as lines where the occurrences are considered to be hypertext lines. The reference associated with each of these lines has the following form: `<a href="X Y Z">`. `X` and `Y` represent the start and end position of the occurrence in characters in the file `name_of_text.snt`. `Z` represents the number of the sentence in which this occurrence appears.

All spaces that are at the left and right edges of lines are encoded as non-breaking space (` ` in HTML), which allows the preservation of the alignment of the utterances even if one of them (one that is at the beginning of the file) has a left context with spaces.

NOTE: if the concordance has been constructed with the parameter `glossanet`, the HTML file obtains the same structure, except for the links. In these concordances, the occurrences are real links pointing at the web server of the GlossaNet application. For more information on GlossaNet, consult the link on the Unitex web site (`http://www-igm.univ-mlv.fr/~{}uni`

Here an example of a file:

```
<html lang=en>¶
<head>¶
¶
   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">¶
   <title>6 matches</title>¶
</head>¶
<body>¶
<table border="0" width="100%"><td nowrap>¶
<font face="Courier new" size=3>¶
on, there <a href="116 124 2">extended</a> i <br>¶
 extended <a href="125 127 2">in</a> ancient <br>¶
 Scott {S}<a href="32 34 2">IN</a> THAT PL <br>¶
STRICT of <a href="61 66 2">merry</a> Engl <br>¶
S}IN THAT <a href="40 48 2">PLEASANT</a> D <br>¶
 which is <a href="84 91 2">watered</a> by <br>¶
</font>¶
</td></table></body>¶
</html>¶
```

Figure 10.2 shows the page that corresponds to the file below.

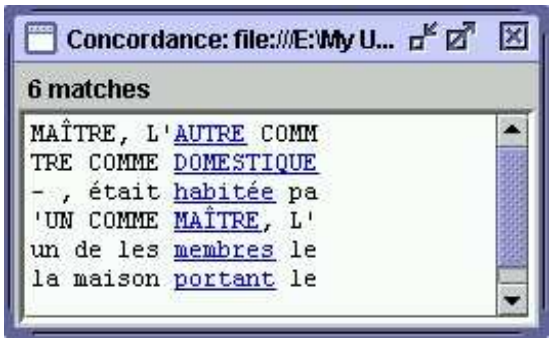

Figure 10.2: Example of a concordance

10.7 Dictionaries

The compression of the DELAF dictionaries done by the program `Compress` produces two files: a `.bin` file that represents the minimal automaton of the inflected forms of the dictionaries, and a `.inf` file that contains the compressed forms allowing the dictionaries to be reconstructed from the inflected forms. This section describes the format of these two file types, as well as the format of the file `CHECK_DIC.TXT`, which contains the result of the verification of a dictionary.

10.7.1 The .bin files

A .bin file is a binary file that represents an automaton. The first 4 bytes of the file represent an number that indicates the size of the file in bytes. The states of the automaton are encoded in the following way:

- the first two bytes indicate if the state is final as well as the number of transitions that leave it. The highest bit is 0 if the state is final, 1 if not. The other 15 bits encode the number of transitions. Example: a non-final state with 17 transitions is encoded by the

  hexadecimal sequence 8011

- if the state is final, the three following bytes encode the index in the `.inf` file of the compressed form to be used to reconstruct the dictionary lines for this inflected form.

  Example: if the state refers to the compressed form with the index 25133, the corresponding hexadecimal sequence is `00622D`

- each outgoing transition is then encoded in 5 bytes. The first 2 bytes encode the character that labels the transition, and the three following encode the byte position of the result state in the `.bin` file. The transitions of a state are encoded next to each other.

  Example: a transition that is labeled with the letter `A` pointing at the state of which the description starts at byte 50106, is represented by the hexadecimal sequence `004100C3BA`.

By convention, the first state of the automaton is the initial state.

10.7.2 The .inf files

A `.inf` file is a text file that describes the compressed files that are associated with a `.bin` file. Here an example of a `.inf` file:

```
0000000006¶
_10\0\0\7.N¶
.PREP¶
_3.PREP¶
.PREP,_3.PREP¶
1-1.N+Hum:mp¶
3er 1.N+AN+Hum:fs¶
```

The first line of the file indicates the number of compressed forms that it contains. Each line can contain one or more compressed forms. If there are multiple forms, they are separated by commas. Each compressed form is made up of a sequence that allows to find a canonical form again starting from an inflected form, followed by a sequence of grammatical, semantic and inflectional codes that are associated with the entry.

The mode of the compression of the canonical form varies with the function of the inflectd form. If the two forms are identical, the compressed form summarizes the grammatical, semantic and inflectionary information like this:

```
.N+Hum:ms
```

If the forms are different, the compression program cuts up the two forms in units. These units can be a space, a hyphen, or a sequence of characters that contain neither a space nor a hyphen. This way of cutting up units allows to efficiently take into account the inflections of the composed words.

If the inflected and the canonical form do not have the same number of units, the the program encodes the canonical form by the number of characters to remove from the inflected form followed by the characters to append. Thus, the first line of the file below corresponds to the line in the dictionary:

```
James Bond,007.N
```

Since the sequence `James Bond` contains three units and `007` only one, the canonical form is encoded with `_10\0\0\7`. The `_` character indicates that the two forms do not have the same number of units. The following number (here 10) indicates the number of characters to remove. The sequence `\0\0\7` indicates that the sequence `007` should be appended. The digits are preceded by the `\` character so they will not be confused with the number of characters to remove.

Whenever the two forms have the same number of units, the units are compressed two by two. If the two units are composed of a space or a hyphen, the compressed form of the unit is the unit itself, as in the following line:

```
1-1.N+Hum:mp
```

This allows to maintain a certain visibility in the `.inf` file whenever the dictionary contains composed words.

Whenever at least one of the units is neither a space nor a hyphen, the compressed form is composed of a number of characters to remove followed by the sequence of characters to append. Thus, the dictionary line:

```
première partie,premier parti.N+AN+Hum:fs
```

is encoded by the line:

```
3er 1.N+AN+Hum:fs
```

The code `3er` indicates that 3 characters are to be removed from the sequence `première` and the characters `er` are to be appended to obtain `premier`. The `1` indicates that only one character needs to be removed from `partie` to obtain `parti`. The number `0` is used whenever it needs to be indicated that no letter should be removed.

10.7.3 The file CHECK_DIC.TXT

This file is produced by the dictionary verification program `CheckDic`. It is a text file that contains information about the analysed dictionary and has four parts.

The first part is the possibly empty list of all syntax errors found in the dictionary: missing of the inflected or the canonical form, the grammatical code, empty lines, etc. Each error is described by the number of line it concerns, a message describing the error, and the contents of the line. Here an example of a message:

```
Line 12451: no point found
garden,N:s
```

The second and third part display the list of grammatical codes and/or semantic and inflectional codes respectively. In order to prevent coding errors, the program reports encodings that contain spaces, tabs, or non-ASCII characters. In addition to that, if a Greek dictionary contains the code `ADV` or the character `A` and the Greek `A` is used instead of the Latin `A`, the program reports the following warning:

```
ADV warning: 1 suspect char (1 non ASCII char): (0391 D V)
```

Non-ASCII characters are indicated by their hexadecimal character number. In the example below, the code `0301` represents the Greek `A`. The spaces are indicated by the sequence `SPACE`:

```
Km s warning: 1 suspect char (1 space): (K m SPACE s)
```

When the following dictionary is verified:

```
1,2 et 3!,.INTJ
abracadabra,INTJ
supercalifragilisticexpialidocious,.INTJ
damned,. INTJ
```

the following file `CHECK_DIC.TXT` is obtained:

```
Line 1: unprotected comma in lemma¶
1,2 et 3!,.INTJ¶
Line 2: no point found ¶
abracadabra,INTJ ¶
---------------------------------- ¶
----  All chars used in forms  ---- ¶
---------------------------------- ¶
  (0020) ¶
! (0021) ¶
, (002C) ¶
```

```
1 (0031) ¶
2 (0032) ¶
3 (0033) ¶
I (0049) ¶
J (004A) ¶
N (004E) ¶
T (0054) ¶
a (0061) ¶
b (0062) ¶
c (0063) ¶
d (0064) ¶
e (0065) ¶
f (0066) ¶
g (0067) ¶
i (0069) ¶
l (006C) ¶
m (006D) ¶
n (006E) ¶
o (006F) ¶
p (0070) ¶
r (0072) ¶
s (0073) ¶
t (0074) ¶
u (0075) ¶
x (0078) ¶
------------------------------------------------------------ ¶
----    2 grammatical/semantic codes used in dictionary  ---- ¶
------------------------------------------------------------ ¶
INTJ ¶
 INTJ warning: 1 suspect char (1 space): (SPACE I N T J) ¶
------------------------------------------------------ ¶
----    0 inflectional code used in dictionary  ----- ¶
------------------------------------------------------ ¶
```

10.8 ELAG Files

10.8.1 The tagset.def file

see section 7.3.6. page XXX

10.8.2 The .lst files

THE .LST FILES ARE NOT ENCODED IN UNICODE !

A `.lst` file contains a list of names of `.grf` iles located in the ELAG directory of the current language. The`elag.lst` file provided for French looks like this:

< get from French version>

10.8.3 The .elg files

The .elg files contain the compiled ELAG rules. These files have the .fst2 format.

10.8.4 The .rul files

These files list the various `.elg` files that represent a set of ELAG rules. A .rul file consist of as many parts as there are .elg files. Each part consists of the list of ELAG grammars that correspond to a .elg file. Each filename is preceded by a tab, followed by a line that contains the `.elg` filename in angle brackets. The lines starting with a tab serve as comments and are ignored by the `Elag` program. The default file `elag.rul` for French looks like this:

< get from French version>

10.9 Configuration files

10.9.1 The file Config

Whenever the user modifies his preferences for a given language, these modifications are saved in a text file named 'Config' which can be found in the directory of the current language. The file has the following syntax:

```
TEXT FONT NAME=Courier New¶
TEXT FONT STYLE=0¶
TEXT FONT SIZE=10¶
CONCORDANCE FONT NAME=Courier new¶
CONCORDANCE FONT HTML SIZE=3¶
INPUT FONT NAME=Times New Roman¶
INPUT FONT STYLE=0¶
INPUT FONT SIZE=10¶
OUTPUT FONT NAME=Times New Roman¶
OUTPUT FONT STYLE=1¶
OUTPUT FONT SIZE=12¶
DATE=true¶
FILE NAME=true¶
PATH NAME=false¶
FRAME=true¶
RIGHT TO LEFT=false¶
BACKGROUND COLOR=16777215¶
FOREGROUND COLOR=0¶
AUXILIARY NODES COLOR=13487565¶
COMMENT NODES COLOR=16711680¶
SELECTED NODES COLOR=255¶
ANTIALIASING=true¶
```

```
HTML VIEWER=¶
MAX TEXT FILE SIZE=1024000¶
ICON BAR POSITION=West¶
```

The first three lines indicate the name, the style and the size of the font used to display texts, dictionaries, lexical units, sentences in text automata, etc.

The parameters `CONCORDANCE FONT NAME` and `CONCORDANCE FONT HTML SIZE` define the name, the size and the font to use when displaying concordances in HTML. The size of the font has a value between 1 and 7.

The parameters `INPUT FONT ...` and `OUTPUT FONT ...` define the name, the style and the size of the fonts used for displaying the paths and the transductions of the graphs.

The following 10 parameters correspond to the parameters given in the headings of the graphs. Table 10.3 describes the correspondences.

Parameters in the `Config` file	Parameters in the `.grf` file
DATE	DDATE
FILE NAME	DFILE
PATH NAME	DDIR
FRAME	DFRAME
RIGHT TO LEFT	DRIG
BACKGROUND COLOR	BCOLOR
FOREGROUND COLOR	FCOLOR
AUXILIARY NODES COLOR	ACOLOR
COMMENT NODES COLOR	SCOLOR
SELECTED NODES COLOR	CCOLOR

Table 10.3: Meaning of the parameters

The parameter `ANTIALIASING` indicates whether the graphs as well as the sentence automata are displayed by default with the antialiasing effect.

The parameter `HTML VIEWER` indicates the name of the browser to use for displaying the concordances. If no browser name is specified, the concordances are displayed in a Unitex window.

10.9.2 The file system_dic.def

The file `system_dic.def` is a text file that describes the list of system dictionaries that are applied by default. This file can be found in the directory of the current language. Each line corresponds to a name of a .bin file. The system dictionaries are in the system directory, and

in that directory in the sub-directory `(current language)/Dela`. Here an example of the file:

```
delacf.bin¶
delaf.bin¶
```

10.9.3 The file user_dic.def

The file `user_dic.def` is a text file that describes the list of dictionaries the user has defined to apply by default. This file is in the directory of the current language and has the same format as the file `system_dic.def`. The dictionaries need to be in the sub-directory `(current language)/Dela` of the personal directory of the user.

10.9.4 The file user.cfg

Under Linux, Unitex expects the personal directory of the user to be called `unitex` and expects it to be in his root directory (`$HOME`). Under Windows, it is not always possible to associate a directory with a user per default. To compensate for that, Unitex creates a `.cfg` file for each user that contains the path to his personal directory. This file is saved under the name `(user login).cfg` in the sub-directory of the system `Unitex/Users`.

ATTENTION: THIS FILE IS NOT IN UNICODE AND THE PATH OF THE PERSONAL DIRECTORY IS NOT FOLLOWED BY A NEWLINE.

10.10 Various other files

For each text Unitex creates multiple files that contain information that is displayed in the graphical interface. This section describes these files.

10.10.1 The files dlf.n, dlc.n et err.n

These three files are text files that are stored in the text directory. They contain the number of lines of the files `dlf`, `dlc` and `err` respectively. These numbers are followed by a newline.

10.10.2 The file stat_dic.n

This file is a text file in the directory of the text. It has three lines that contain the number of lines of the files `dlf`, `dlc`, and `err`.

10.10.3 The file stats.n

This file is in the text directory and contains a line in the following form:

```
3949 sentence delimiters, 169394 (9428 diff) tokens, 73788 (9399) simple
 forms, 438 (10) digits
```

The numbers indicated are interpreted in the following way:

- `sentence delimiters`: number of sentence separators (`{S}`);
- `tokens`: total number of lexical units in the text. The number preceding `diff` indicates the number of different units;
- `simple forms`: the total number of lexical units in the text that are composed of letters. The number in parentheses represents the number of different lexical units that are composed of letters;
- `digits`: the total number of digits used in the text. The number in parentheses indicates the number of different digits used (10 at the most).

10.10.4 The file concord.n

The file `concord.n` is a text file in the directory of the text. It contains information on the last search done on the text and looks like the following:

```
6 matches¶
6 recognized units¶
(0.004% of the text is covered)¶
```

The first line gives the number of found occurrences, and the second the number of units covered by these occurrences. The third line indicates the ratio between the covered units and the total number of units in the text.

Index

Bibliography

[1] M. CONSTANT, T. NAKAMURA, and S. PAUMIER. L'héritage des gènes MG. la localisation des auxiliaires en français. Actes du 21e colloque international Grammaires et Lexiques Comparés.

[2] Sébastien PAUMIER. Nouvelles méthodes pour la recherche d'expressions dans de grands corpus. In Anne Dister, editor, *Revue Informatique et Statistique dans les Sciences Humaines*, volume Actes des 3èmes Journées INTEX, pages 289–295, 2000.

[3] Sébastien PAUMIER. Recherche d'expressions dans de grands corpus : le système AGLAE, 2000. Mémoire de DEA.

[4] Sébastien PAUMIER. Some remarks on the application of a lexicon-grammar. In *Lingvisticæ Investigationes*, number 24, Amsterdam-Philadelphia, 2001. John Benjamins Publishing Company.

[5] Sébastien PAUMIER. Some remarks on the application of a lexicon-grammar. http://www.nyu.edu/pages/linguistics/intex/downloads/Sebastien%20Paumier.pd 2001. Online Proceedings of the 4th Intex workshop.

[6] Sébastien PAUMIER. UNITEX - manuel d'utilisation. http://www-igm.univ-mlv.fr/~unitex/manuelunitex.ps, 2002.