INTEX

Max Silberztein, Université de Franche-Comté

Translation by Michael Long, Université de Moncton

TABLE OF CONTENT

I. Getting Started

This first part, "**Getting Started**", introduces you to INTEX (chapter 1), then shows you how to get started with the software (chapter 2).

Chapter 1. WELCOME

1.1. Introduction

INTEX is a development environment used to construct large-coverage formalized descriptions of natural languages, and apply them to large texts in real time. The descriptions of natural languages are formalized as electronic dictionaries, as grammars represented by finite state graphs and as lexicon-grammars.

INTEX supplies tools to describe inflectional and derivational morphology, terminological and spelling variations, vocabulary (simple words, compound words and frozen expressions), semi-frozen phenomena between lexicon and syntax (local grammars) and syntax (phrase and sentence grammars).

INTEX is also used as a corpus processing system: it allows one to process texts as large as several hundreds of megabytes in real time (typically, the equivalent of 150 pocket novels). Typical operations include indexing morpho-syntactic patterns, frozen or semi-frozen expressions (technical expressions, for example), lemmatized concordances, and the statistical study of the results.

1.2. System requirements

INTEX functions in Windows 95-98-ME, Windows NT-2000 and Windows XP. Certain initial versions of these systems contain malfunctions (bugs) that may affect the smooth functioning of the system. We strongly advise that you update your operating system, particularly downloading the latest “Service Pack”.

The minimum requirements for a computer to run INTEX on texts the size of a novel (around 1 Mb) are not very high: Pentium 3+-type PC with 64 Mb of RAM, 500 Mb available on the hard drive, 17 inch screen, 1024x768 16-bit resolution with a minimum of 75 Hz refresh rate.

If INTEX is to be used for the analysis of large documents (50 Mb or more), or if INTEX is used to compile large-coverage dictionaries (10,000 or more entries), or lexicon grammar tables, we advise the following minimum configuration: PC with Pentium 3+, 256 MB RAM, 2 GB available space on hard drive.

If INTEX is used as a development tool to build graphs, a good screen is necessary: at least a 19 inch screen, 1600x1024 16-bit resolution with a minimum of 80 Hz refresh rate.

1.3. Tools

Finite-State Automata (FSA)

For INTEX, **Finite-State Automata** are a special case of finite-state transducers, that do not produce any information (i.e. with no output) other than the binary information “sequence is recognized”, or “sequence is not recognized”. Typically, one will use finite-state automata to extract (search, index, extract, count, etc…) certain sequences of interest in texts.

Regular Expressions

Regular expressions constitute a quick way to enter simple finite-state automata, without having to build graphs. When the sequence to be located consists of one, two, or three words, it is much quicker to enter these words directly into a regular expression; however, if the structure to be located becomes more complex, one should build a graph.

Finite-State Transducers (FSTs)

One of INTEX's essential characteristics is that most of the objects processed (texts, dictionaries, grammars) are, at some point, represented by **finite-state transducers**, or variants of them.

A finite-state transducer (FST) is a graph that represents a group of incoming recognized sequences, and associate them with a group of outgoing produced sequences. Typically, a grammar will represent word sequences (read in the text), and produce linguistic information (information on the syntactic structure for example); a dictionary will represent sequences of letters (that spell each lexical entry), and produce lexical information (part of speech, inflection codes,…); the transducer of a text will represent the word sequences (that form each sentence) and assign them lexical and/or syntactic information (the linguistic markers produced by the different analyses).

Representing these three objects in the same way presents considerable advantages, especially in terms of speed of execution. All of the operations carried out by INTEX can in fact be expressed by a limited number of operations carried out by finite-state transducers. For example, applying dictionaries to a text will consist in building the union of these dictionaries' transducers (the result is also a transducer), and projecting this transducer on the text's transducer. Furthermore, INTEX offers completely new operations such as applying grammars to dictionaries (to verify their format, for example).

Enhanced Finite-State Transducers (EFSTs)

Enhanced Finite State Tranducers are FSTs that contain variables; these variables are set during parsing, as they can store affixes of matching sequences; their content can then be copied to the output part of the EFST, in order to perform powerful modifications in the text.

Recursive Transition Networks (RTNs)

Recursive Transition Networks are INTEX graphs that contain one or more references to embedded graphs; these latter graphs may in turn contain other references, to the same, or other graphs. Generally, RTNs are used in INTEX to build libraries of graphs from the bottom-up: simple graphs are designed, then re-used by more and more general graphs.

Context-Free Grammars (CFGs)

For INTEX, **Context-Free Grammars** are a special case of Recursive Transition Networks that do not produce any information (i.e. with no output), other than the binary information “sequence is recognized”, or “sequence is not recognized”.

Enhanced Recursive Transition Networks (ERTNs)

Enhanced Recursive Transition Networks are RTNs that contain variables; one uses these variables to store affixes of a matching sequence, perform some operation with them, and then insert them in the output of the graph.

INTEX Commands

The functions available via the INTEX user interface are also available as Windows standalone programs, or commands. One can therefore build applications that are as powerful as INTEX simply through calling the commands, either in a "SHELL" script or in more sophisticated programs written in PERL, C++, etc.

This feature also allows users to modify the INTEX behavior: for instance, a user can replace one or more INTEX commands with his/her own. In that case, the INTEX user interface can be used to launch other tools.

1.4. Linguistic Resources

With INTEX, linguists can design, maintain and test three types of linguistic resources:

-- DELA-type dictionaries are morpho-syntactic dictionaries; these dictionaries usually associate simple or compound word forms with their morpho-syntactic categories (i.e. "Verb"), optional syntactic features (i.e. "+transitive") and distributional classes (i.e. "+Human"), and some inflectional information (i.e. "third person singular, Present").

-- INTEX graphs are used to represent a large gammut of linguistic phenomena, from the orthographical and the morphological level, up to the syntagmatic and transformational syntactic level.

-- Lexicon-Grammar tables are data bases that describe linguitic phenomena that are at the borderline between lexicon and syntax. For instance, frozen expressions and associations (Support Verb / Predicative Noun), in a way, belong to the vocabulary of a language, and thus should be described in a lexicon; on the other hand, their structure, and their syntactic properties are as general as free phrases and full sentences.

1.5. The INTEX community

To this day, more than 200 users, in a dozen countries, are using INTEX as a research or educational tool. Some of them are interested by its Corpus processing fuctionalities (literary text analysis, researching information in newspapers or technical documents, etc…), others are using it as a platform to formalize certain phenomena in linguistics (e.g. describing a language's morphology, lexicon, and expressions), others for its computational power (automatic text analysis).

These users make up a real community, and we greatly encourage the new users to join the info-intex mailing list as well as the annual INTEX workshop.

1.6. Structure of the book

This book is divided into five parts:

(1) This first section "**Getting Started**" presents INTEX (chap.1), takes you through the installation process and gives you the minimum amount of information necessary to launch a basic search in a text (chap. 2);

(2) The second section "**Regular expressions and graphs**" shows you how to carry out simple searches in texts with regular expressions (chap. 3), how to use lexical resources for linguistic queries (chap. 4), and how to use INTEX's graph editor to describe more complex queries (chap.5);

(3) The third section "**INTEX Grammars**" presents local grammars, and how to build libraries of graphs in order to build a bottom-up description of Natural languages (chap. 6), then Finite-State Transducers, and how to use them to perform modifications on texts (chap. 7), then more powerful grammars, such as Context-Free grammars, Enhanced FSTs and RTNs (chap. 8);

(4) The fourth section "**Text Processing**" (chap. 9 & 10) explains how to prepare texts to be used with INTEX (chap. 9). From a "raw" text in a Windows ANSI format, you will apply transducers that will segment the text into sentences and then standardize the text from a linguistic point of view (chap. 10);

(5) The fifth section "**Lexical analysis**" (chap. 11 and 12) describes the DELA simple word and compound word dictionaries as well as lexical graphs (chap. 11); you will also learn to construct your own tokenizers and morphological graphs (chap. 12);

(6) The sixth section "**Syntactic analysis**" (chap.13 & 14) details the disambiguation module of INTEX (chap. 13), and the INTEX syntactic parser (chap. 14);

(7) The seventh section "**Advanced Lexicons**" presents INTEX tools that can help build DELA-type dictionaries (chap. 15) and lexicon-grammar tables (chap. 16);

(8) The last section "**INTEX for developers**" describes each of the 30+ INTEX commands that are used by the INTEX application (Chap. 17); these commands are standalone programs that can also be directly used from a command-line "DOS" windows or a UNIX Shell environment. Chapter 18 describes the files, directories and file formats used by INTEX and these programs. Finally Chapter 19 gives a non-exhaustive reference on related works.

Chapter 2. INSTALLING THE SOFTWARE

2.1. Installing the software

-- If you have the INTEX CD-ROM, put it in your CD-ROM drive and wait a few moments. If your system automatically detects the new CD, the installation program will launch automatically; move on to the next sub-section for the actual installation. If nothing happens, double-click on the *My Computer* icon in order to view the following window:

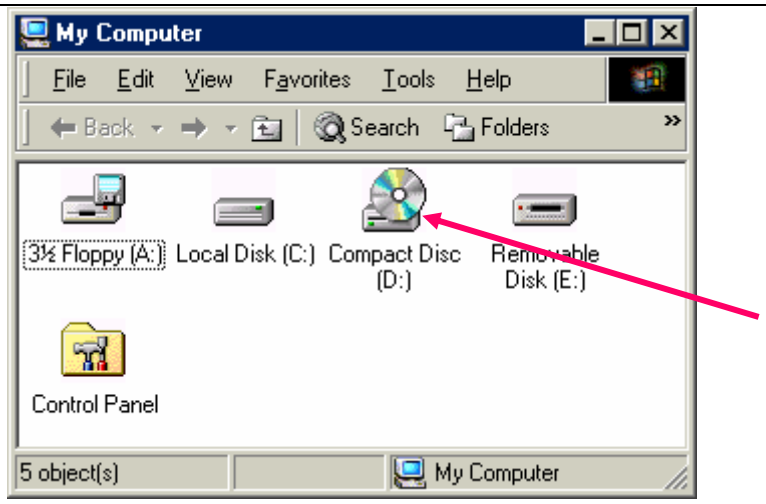


Figure 1. *Double-click on the CD-ROM Drive (D:)*

Double-click on the CD-ROM icon, then double-click on the file **Setup.exe** to launch the installation.

If the extension of the files' names is not displayed, you will see three files named **Setup.** In general, we advise that you set the Windows interface to display filename extensions because, in many cases, different INTEX files share the same name but have different extensions, each extension noting a different file format (e.g. ".txt" for the Windows ANSI version of a text; ".snt" for the segmented INTEX version).

-- You can also download INTEX from the INTEX Web sites: www.univ-fcomte.fr/laseldi/intex; you get to dowload the file "intex.zip"; uncompress it; locate the file "Setup.exe", double-click on it to launch the installation.

IMPORTANT: Before proceeding any further (**before** the software registration), make sure that you have the latest INTEX upgrade. Point your browser to the INTEX website : www.univ-fcomte.fr/laseldi/intex. There, click on the latest upgrade button, and run the Upgrade.exe file.

Follow the instructions that pop up on your screen. We recommend that you choose the default options.

When you are done installing the INTEX files, you can choose between running the INTEX program directly, or running it by clicking on **Start** in the Windows taskbar, then **Intex**, then **Intex** once again.

2.2. Registering the software, uncrypting the data

The first time you run INTEX, the registration window will appear. Carefully write down your "Machine Identification Number": this number will allow us to compute the installation key needed to uncrypt the dictionaries, and activate the INTEX programs on your machine. You will need the following information: a contact name, e.g. "John Smith", an institution name, e.g. "New York University".

You can obtain a licence number and the installation key by sending an e-mail with your contact and institution name to: max.silberztein@univ-fcomte.fr.

After having entered the installation key, the **Completing Installation** window will appear; click on **OK**: the INTEX dictionaries are being uncrypted and the programs are activated. Make sure you leave enough time for the computer to uncrypt all files; you should wait for the message "Installation successful; please relaunch INTEX". After the unlocking, INTEX is ready to be used.

2.3. Creating your personal folder

The **User folder** is the folder within which INTEX will store your personal data: on the one hand, your texts, your dictionaries, and your grammars, and on the other the results of your processing (index, concordances, etc…)

If you are the only INTEX user working on your computer, skip this section; the INTEX user folder created by default is **C:\Intex**.

However, if several users will be working on the same machine, each user wants to be able to work on different texts, to edit his/her own versions of the standard dictionaries, or to apply different grammars or dictionaries to the same text. Therefore we recommend that each user creates one different user folder, such as: **C:\Users\Max\Intex, C:\Users\Nancy\Intex**, etc…

NOTE to Windows NT/2000/XP users: if you have installed INTEX under the administrator account, it is better to close the session, then log on with your personal account in order to create your User folder. Indeed, if your user folder is created by the privileged Administrator, you may not have the right to modify your own files.

The folder may be created just about anywhere. If your hard drive has several partitions, or if you have several hard drives, you should naturally create the folder on the drive with the necessary space, typically 100 Mb.

To create the user folder **C:\My Intex**, double click on **My Computer**, then on the **C** drive, then select the menu item **File**, then **New,** then **Folder**. Rename the folder **New Folder** which you just created by typing in **My Intex**, then press **Enter**.

After having created your User folder, you must indicate its location to INTEX. Simply launch INTEX: on the Windows taskbar (usually at the bottom of the screen), click on **Start,** then **Programs,** then **Intex**, then **Intex** again.

Then, go to the **Info** menu, select **Preferences** (the window within which you will select the default INTEX parameters). A window similar to the following will appear:

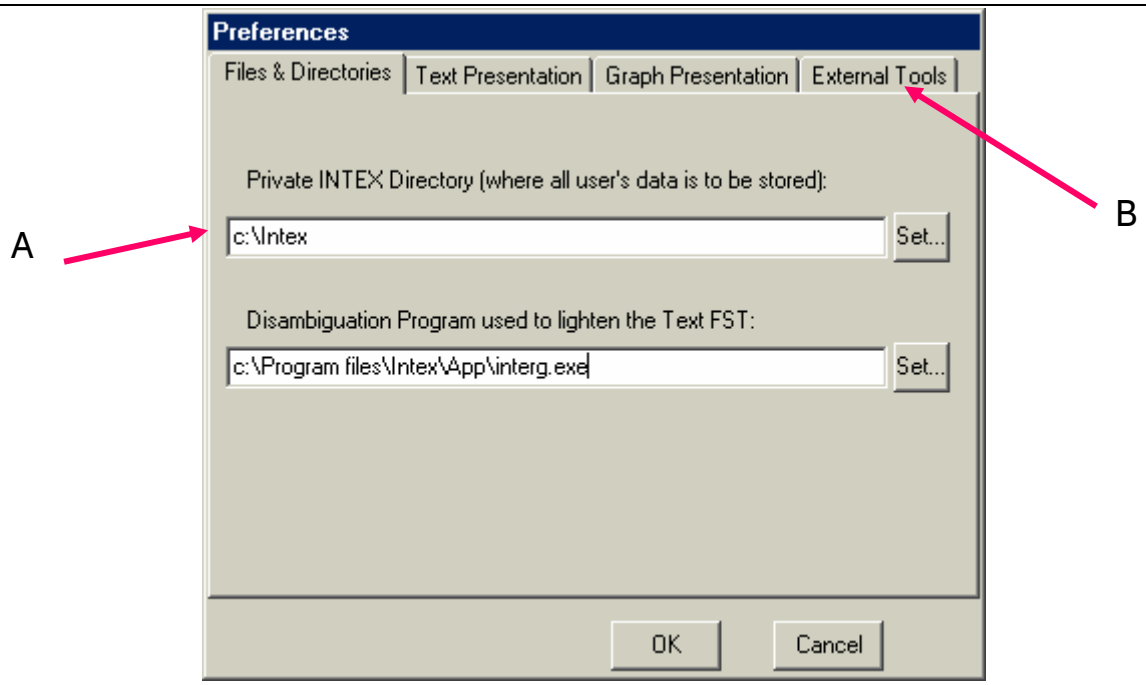


Figure 2. INTEX Personal user folder

(A) You may either type the name of your user folder in the first field: **Private INTEX Directory,** for example: **C:\Max\MyIntex** or **C:My Intex**, or click on the **Set** button, to the right of the field, and find your personal folder in the folder directory.

(B) Use this opportunity to tell INTEX what external tools you want to use. In the Preference window, select the “External tools” tab.

Tell INTEX what text editor will be used to edit dictionaries (notepad in this case in the first field), as well as what word processor used to visualize enriched texts and

concordances (eg. Microsoft Word in the third field). Don't forget to validate your choice by clicking on **OK** at the bottom of the window.

Congratulations: you have installed INTEX; you are now ready to get to work.

2.4. Uninstalling the software

If you wish to uninstall the software, you MUST follow the standard Window method. Be careful NEVER to simply manually delete INTEX files or folders without going through the operating system's assistance.

You must click on the Windows taskbar: **Start,** then **Settings**, then **Control Panel**, double-click on the **Add/Remove Programs;** select **INTEX**, then click on **Uninstall**.

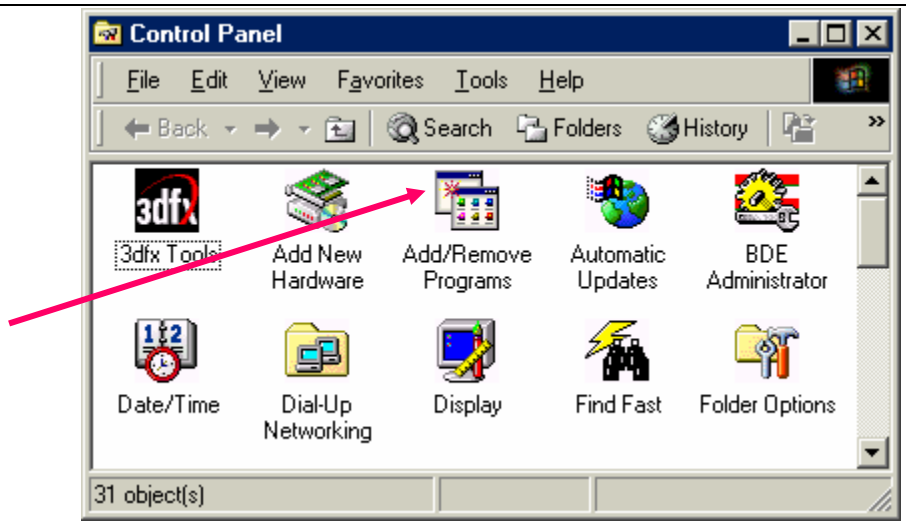


Figure 3. Windows control panel

Since your INTEX data is stored in your user folder (not in the INTEX system folder), they will not be deleted. If you choose to, you can manually delete your user folder by using the standard Windows operation.

Summary: you have installed the software, directed INTEX to your personal user folder which will contain all your data. You also learned to uninstall the software.

We will now explore one of INTEX's fundamental functionalities: the ability to locate words and expressions in a text.

2.5. Loading a text

Launch INTEX (In the Windows taskbar: **Start -> Programs -> Intex -> Intex**). The first window that will appear will allow you to set the working language. For the moment, set **English** as the working language, then click on **OK** to confirm your selection.

Warning: not to confuse the **file** “Portrait of a lady.snt” with the **folder** “Portrait of a lady_ snt”. Note the underscore instead of the dot. For each text represented in a file, INTEX associates a folder within which are stored that text's index, dictionaries, concordances, as well as all of the results based on the processing of that text. The folder has the same name as the text, but its extension dot is replaced with the “_” character (underscore).

First of all, a few definitions:

Letters are the elements of the alphabet of the current language. These characters must actually be listed in the Alphabet file stored in the directory of the current language (the one you select when you launch INTEX). Digits are the ten arabic digits (from “0” to “9”). The Blank in INTEX represents any sequence of spaces, tabulation characters, NEWLINE and CARRIAGE RETURN. Delimiters are all the other characters (i.e. that are neither a letter, a blank or a digit).

From these definitions, INTEX uses the following definitions:

The tokens are the basic objects processed by INTEX. They are classified into four types:
--the simple forms are sequences of letters between two delimiters;
--the tags represent linguistic data, and are noted between brackets “{“ and “}”;
--the digits
--the delimiters.

Note that digits and delimiters are both considered as characters and as tokens.

Some unusual examples:

For INTEX, the sequence “o’clock” is made up of three tokens: the **simple form** “o”, followed by the **delimiter** “ ' “, followed by the **simple form** “clock”; similarly the compound adverb “**a priori**” is made up of two **simple forms** (blanks do not count).

The sequence “3.14” is made up of one **digit**, one **delimiter**, and then two **digits**, that makes four tokens; the sequence “…” is made up of three **delimiters**, hence three

tokens. The sequence "PDP/11" is made up of the **simple form** "PDP", followed by the **delimiter** "/", followed by two **digits** (which makes four **tokens**).

Now click on the **Text** menu, then **Open...** You will see the contents of the **Corpus** folder. Select the file "**Portrait of a lady.snt**" (the novel by Henry James). The text will load and you should see a windows like the one below:

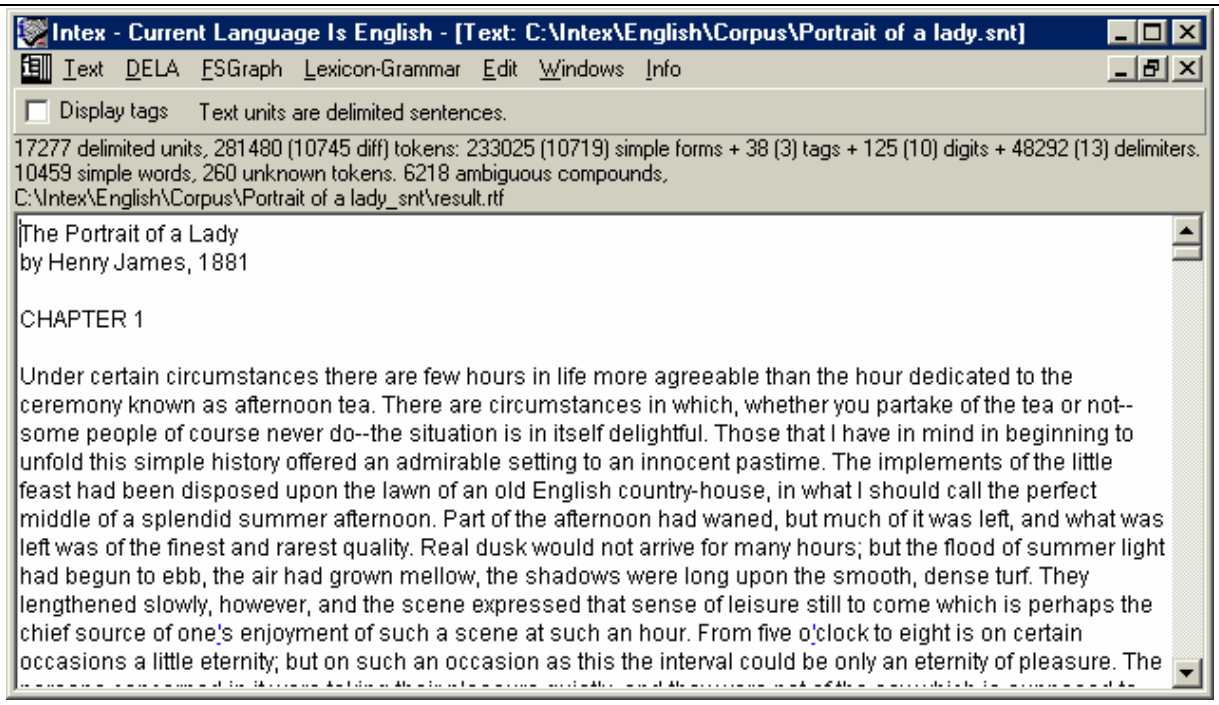


Figure 4. Loading the text "The Portrait of a lady"

Note that INTEX gives some statistical indications above the text window. In the first line, INTEX gives some *formal information* on the text:

- **17,277 delimited units**: the text was divided into 17,277 sentences;
- **281,480 (10,745) tokens**: the text contains 281,480 tokens, of which 10,745 are different ones; these tokens are classified into four types:
- **233,025 (10,719) simple forms**: 233,025 simple forms, (10,719 different ones);
- **38 (3) tags**: 38 tags (3 different tags);
- **125 (10) digits**: 125 digits, (10 different ones, i.e. all ten digits were used);
- **48,292 (13) delimiters**: 48,292 delimiters (i.e. 11 different ones).

From the definition of the tokens, INTEX defines **linguistic units**.

INTEX processes four types of linguistic units:
-- Affixes (prefix, proper affix or suffix) are sequences of letters included in simple forms, that are associated with relevant linguistic data, e.g. *re-*, *-ization*. They are usually used in morphological graphs;
-- simple words are simple forms that are associated with relevant linguistic information, e.g. *table*; they are usually described in dictionaries;

-- compounds are sequences of simple forms (separated by delimiters) associated with relevant linguistic information, e.g. *as a matter of fact* (usually stored in dictionaries);
-- frozen expressions are potentially discontinued sequences of simple forms that are associated with relevant linguistic information, e.g. *take ... into account* (usually described in lexicon-grammars).

In the second line, INTEX gives some *linguistic information* on the text vocabulary:

- **10,459 simple words**
- **260 unknown simple forms**
- **6,218 ambiguous compounds**

Be careful with the interpretation of the data concerning the text's vocabulary:

- "10,459 simple words" means that the 10,719 different simple forms correspond to 10,459 entries in the dictionaries for simple words that were applied to the text.
- "260 unknown tokens" means that 276 different simple forms do not correspond to any lexical entry in any of the simple word dictionaries that were applied to the text. This number, which naturally varies depending on the selection of dictionaries, generally represents proper nouns, foreign words, or typos;
- "6,218 ambiguous compounds" means that 6,218 different sequences of simple forms correspond to an entry in the ambiguous compound dictionaries that ere applied to the text. This does not necessarily mean that there are 6,218 compound words in the text's vocabulary; for example, the compound noun "red tape" (meaning bureaucratic process) would be counted in the following text as an "ambiguous compound", even though it does not occur in fact:

```
You can buy some blue and red tape in this shop.
```

At this stage, no syntactic nor semantic analysis has been performed; the vocabulary of the text is an indication of the linguistic information that potentially will be needed by the next stage of analysis.

Note: The two different **simple forms** (or **tokens**) "THE" and "the" are associated with a unique **simple word**, because there is only one lexical entry "the, Determiner" in the dictionary that matches both simple forms. "**BUSH**", "**Bush**", and "**bush**" are three different simple forms (or tokens) for INTEX; after INTEX consults the system's dictionaries, it can link these three tokens to the simple word "*a bush*" (noun), but only the first two tokens to the proper name (if one has applied a dictionary of proper names). Therefore, the three tokens correspond to two simple forms.

Find the Window "**Tokens List**" which should be minimized at the bottom of the screen (under the text window). Double-click on it and you should see:

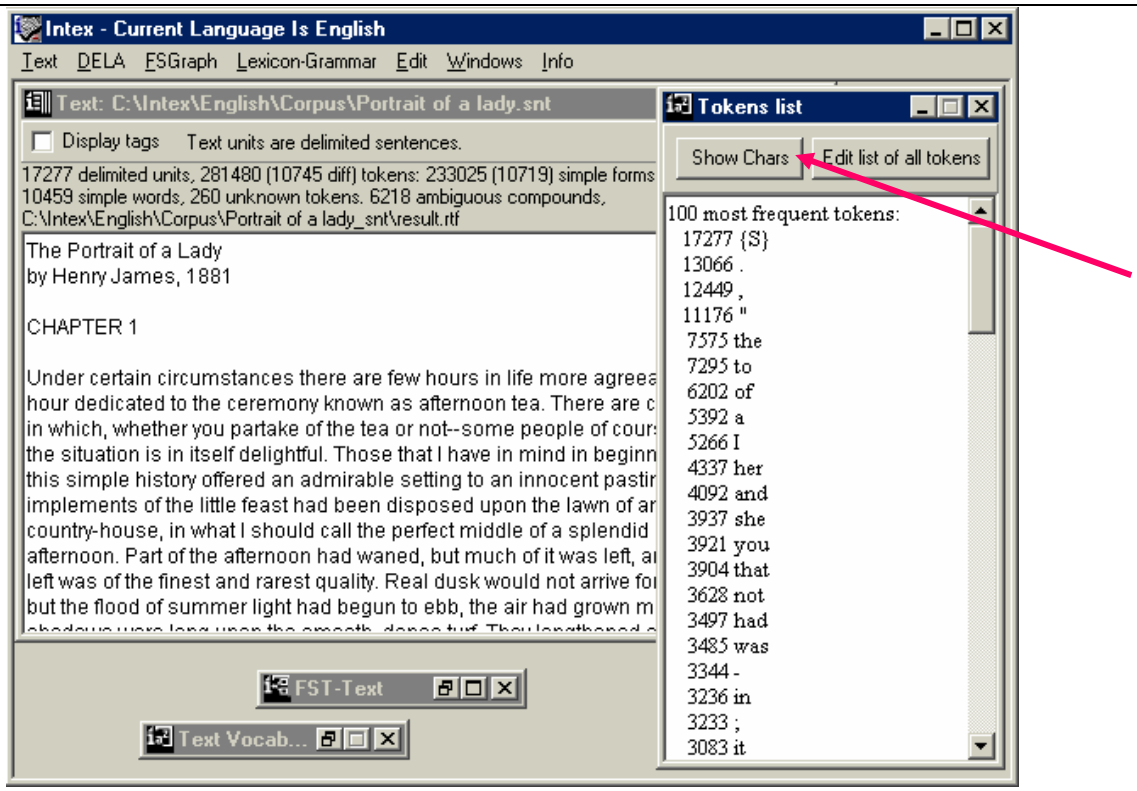


Figure 5. List of the 100 most frequent tokens

The most frequent token in this text is the tag "`{S}`" used to separate sentences of the text, then the dot, which appears 13,066 times, then the comma, the double quote, then the simple form "`the`", etc.

By default, INTEX only lists the 100 most common tokens. By clicking on the button "**Edit list of all tokens**", you may display the entire list of tokens in the text, each token associated with its frequency in the text. By clicking on the button "**Show Chars**", you display the entire list of the text's characters with their Windows ANSI code and their frequency.

2.6. Locating a word

In the **Text** menu, click on "**Locate Pattern…**". The "**Locate Panel**" window will appear. In the field "**Locate pattern in the form of:**", select "**Reg. Expression:**" (you will enter a regular expression), then type "perhaps" in the field (A). Then click **Start** in the lower right corner (B) of the window. The operation is launched.

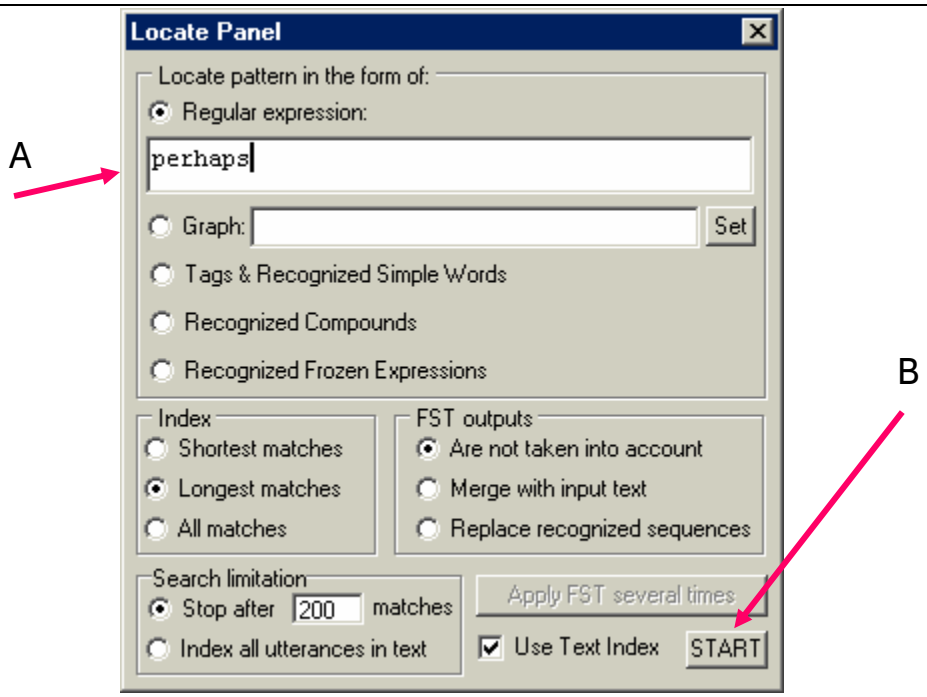


Figure 6. Locate a word

INTEX should quickly let you know that 160 utterances were found. Click on **OK**, then scroll in the text to locate the utterances, which should be underlined in blue.

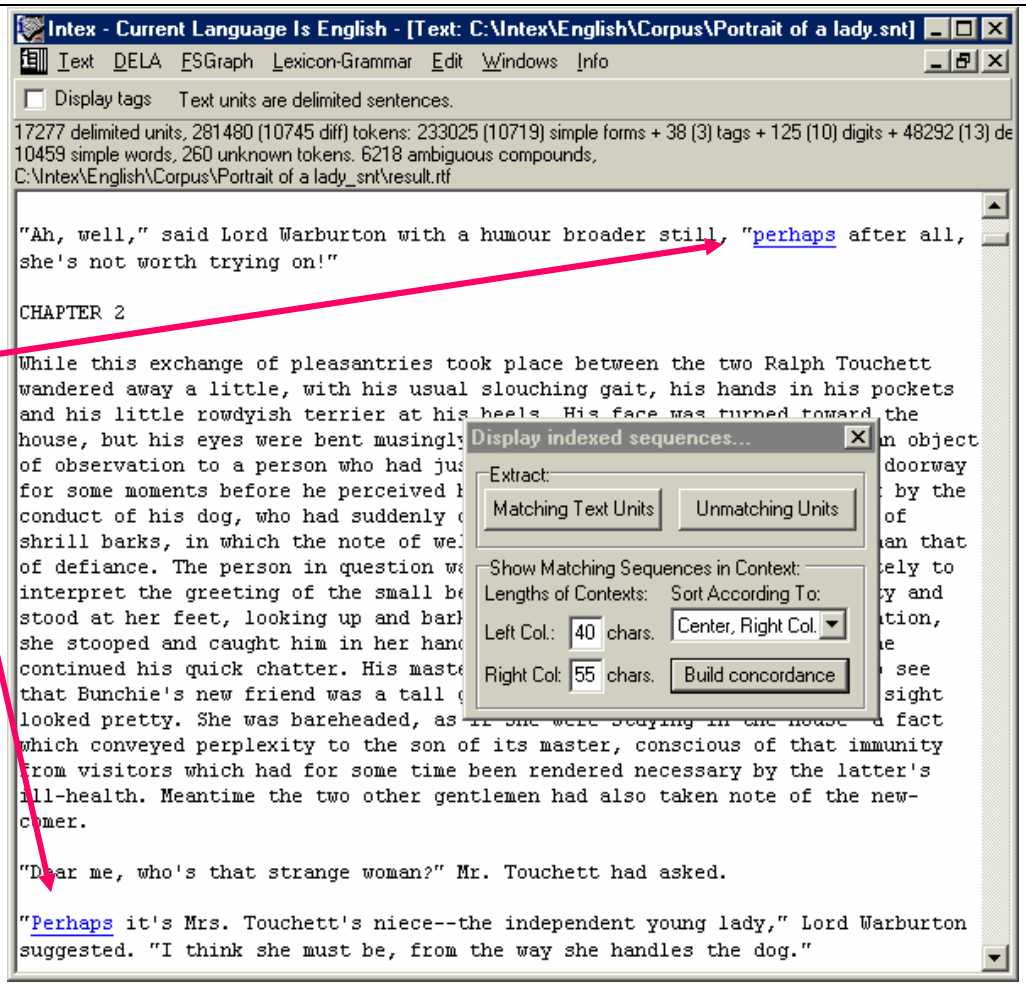


Figure 7. The recognized words are underlined in the text

Browsing the text to find utterances of a word or a sequence is suitable when the text is small and the number of matches is large. However, if there are few matches found in large text (e.g. three matches in a ten thousand page text), it may become difficult to locate them.

The window "**Display indexed sequences**" then may prove more useful: click on "**Build concordance**" to display the concordance of the word "perhaps".

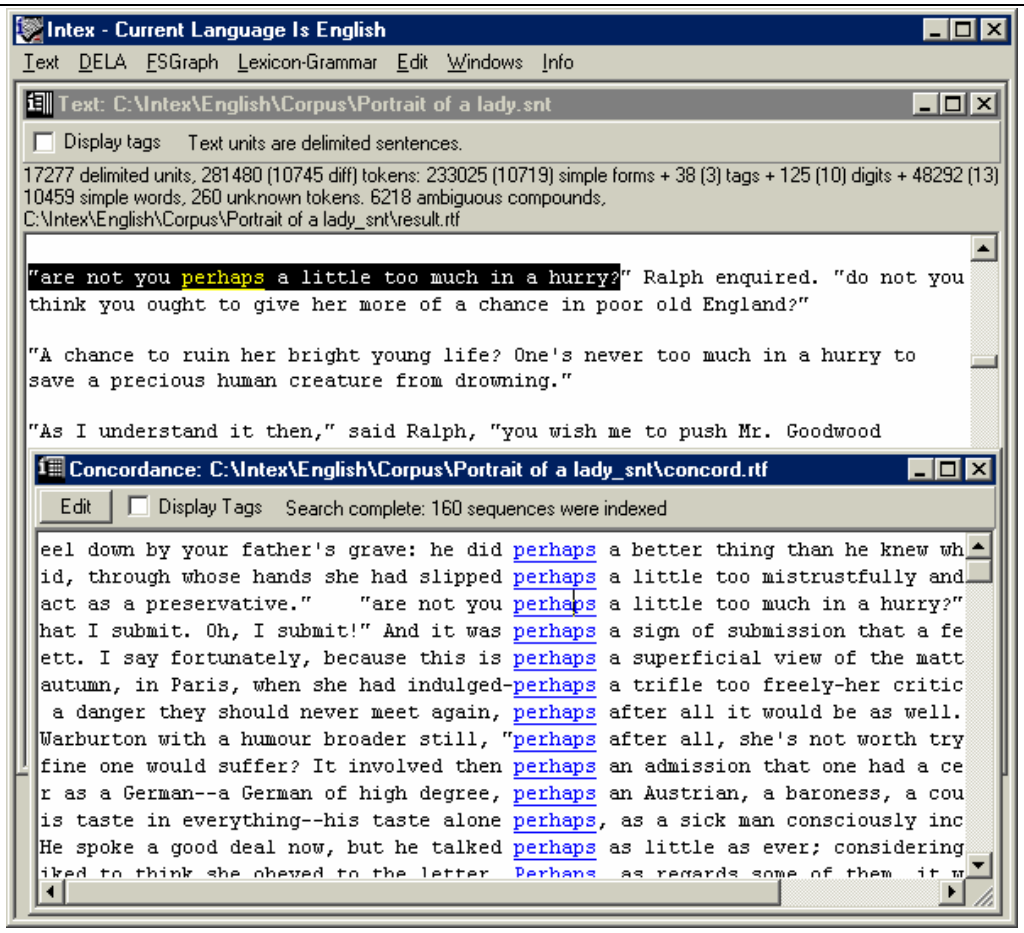


Figure 8. *Concordance of the word "perhaps"*

Definition: The **concordance** of a sequence is an index that represents all of its utterances in context; INTEX concordances are displayed in three columns: each occurrence being presented in the middle column, between its left and its right context.

You can vary the size of the left and right context, as well as the order in which the concordance is sorted. The cursor (generally an arrow) becomes a hand when it hovers above the concordance; if you click on a match and the text window is open, INTEX displays the matching occurrence within the text.

II. Regular expressions and graphs

The second part shows you how to carry out complex searches in texts with regular expressions (chap. 3), how to use lexical resources for linguistic queries (chap. 4), and how to use INTEX's graph editor to describe more powerful queries (chap.5);

Chapter 3. REGULAR EXPRESSIONS

3.1. Disjunction

Reactivate the locate window (**Text -> Locate pattern…,**or use the shortcut **Ctrl+L**). Now type the following **regular expression** (spaces are optional)**:**

```
never + perhaps
```

In INTEX, the disjunction operator (also known as the UNION, or the “or”) is symbolized by the "+" character.

Make sure there is no limitation to the search: select “**Index all utterances in text**” at the bottom of the **Locate Panel**. Since these adverbs are very frequent, we are expecting a high number of matches. If we had left the option "**Stop after 200 matches**", the search would have been limited to the first 200 matches. Now click on **Start**: the search is launched.

When it is finished, construct the concordance of these two words by clicking "**Build concordance**" in the "**Display located sequences**".

Note: the disjunction operator, introduced in INTEX as the character "+", tells INTEX to locate all of the utterances for "never" **or** "perhaps" in the text.

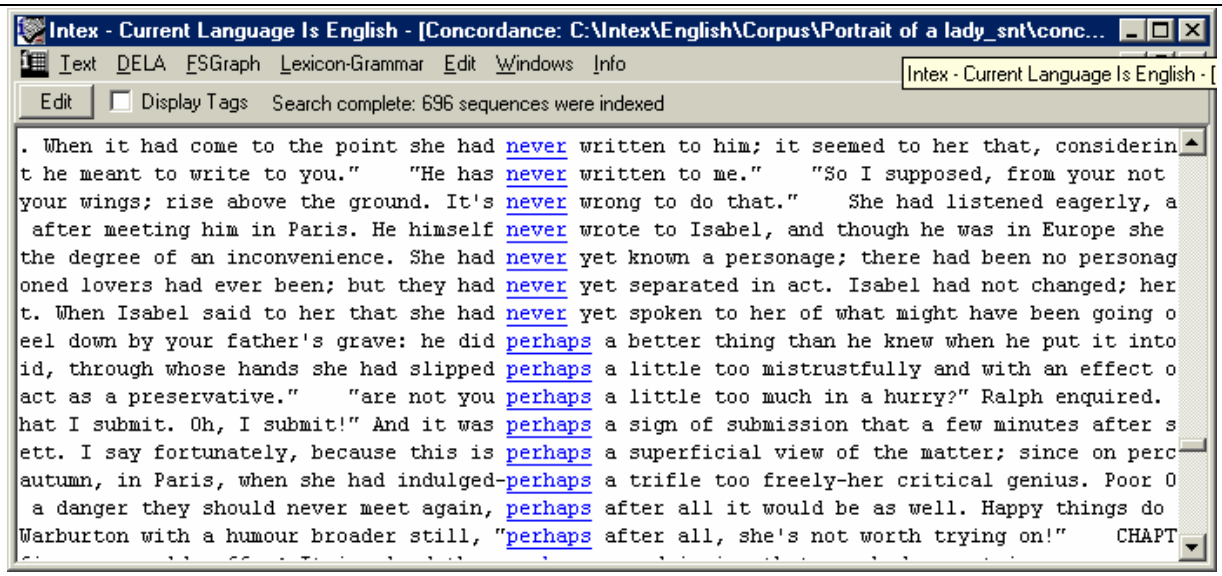


Figure 9. Concordance for the expression: never+perhaps

3.2. Parentheses

We want to locate the sequences made up of the word "her" or "his", followed by the word "voice". To do this, display the locate window (**Text -> Locate pattern**), then enter the following regular expression:

```
(her + his) voice
```

Click on **Start**, then construct the corresponding concordance. INTEX found 19 occurrences of the sequence, *her voice* or *his voice*. Launch the search once more but this time do not use any parentheses:

```
her + his voice
```

This time, INTEX recognized 4,495 utterances:

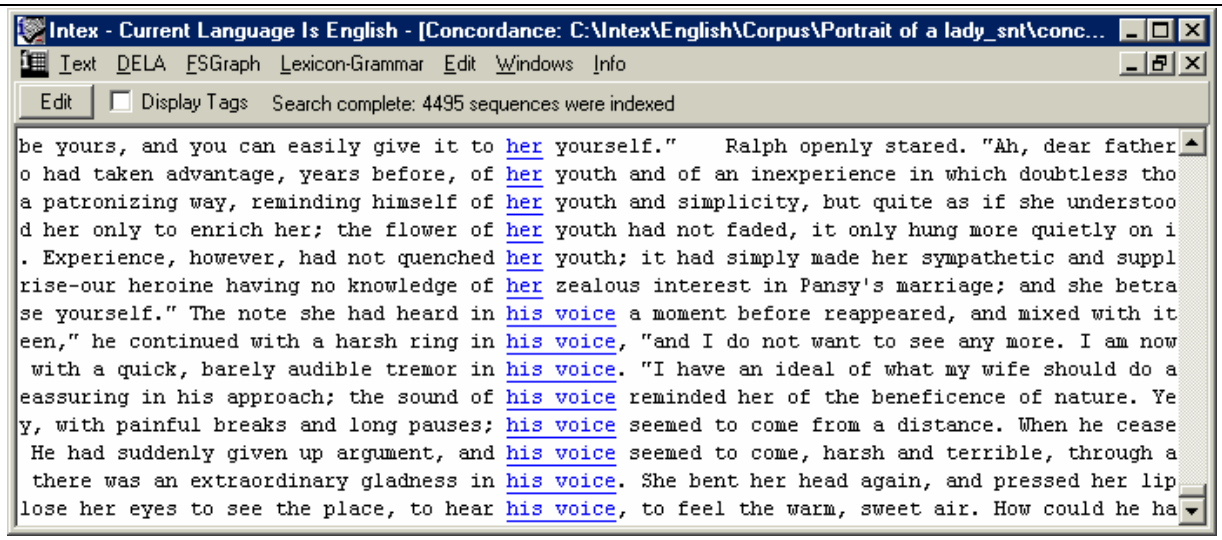


Figure 10. Forgotten parentheses

What happened? INTEX has indexed two sequences: “her” and “his voice”. The blank space, called a concatenation operator, used here between the words *his* and *voice*, takes priority over the "or" operator "+".

In the former regular expression, the parentheses were used to modify the order of priorities, so that the scope of the “or” (the disjunction operator) be limited to *her* or *his*.

In regular expressions, blanks (also named concatenation operators), have priority over the disjunction operator. Parentheses are used to modify the order of priority.

3.3. Sets of forms

We will now locate all of the utterances for the verb *to be*. In the **Text** menu, click on **Locate pattern…** to reload the locate window. Select the option "**Reg. Expression**", then type in (A):

```
am+are+is+was+were
```

In the lower left hand corner (B), under **Search limitation,** make sure the radio button "**Index all utterances in text**" is selected, then click on **Start** to launch the search.

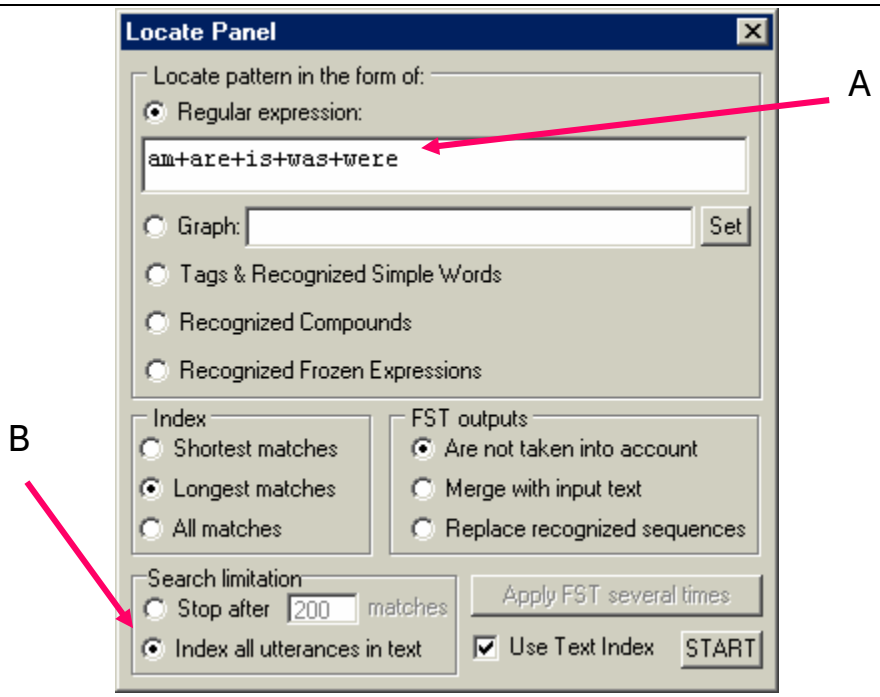


Figure 11. Locate a set of forms

Then, if you wish, you can display the concordance (**Build concordance -> Display indexed Sequences**).

The disjunction operator allows you to undertake several searches at a time; in this example, the forms are all inflectional forms of the same word, but one could also locate spelling variations, such as:

```
csar + czar + tsar + tzar
```

names and their variations, such as:

```
New York City + Big apple + the city
```

terminological variants:

```
camcorder + video camera
```

morphologically derived forms:

```
Stalin + stalinist + stalinism + destalinization
```

Or expressions, terms or forms that represent similar concepts:

```
(credit + debit + ATM + visa) Card + Mastercard
```

Disjunctions therefore turn regular expressions into a powerful tool to extract information from texts.

3.4. Using lower-case and upper-case in regular expressions

In a regular expression, a word written in lower-case recognizes all of its variations in a text. The following expression, for example:

```
it
```

also recognizes the four simple forms:

```
IT, It, it, iT
```

On the other hand, a form that contains at least one upper-case letter in a regular expression will only recognize identical forms in texts; for example:

```
It
```

will recognize **only** the form "It". If you want to recognize the form "it" only when it is written in lower-case, use the quotation marks:

```
"it"
```

will recognize **only** the form "it".

3.5. Exercises

Study the use of the word *girl* in the novel "The portrait of a lady". How many times this word is used in the plural? in how many different compound nouns this form occurs?

How many times the form *death* occurs in the text; in how many idiomatic or metaphoric expressions?

Study the use of the preposition *into*: how many times this preposition has a locative function?

Locate in the text all occurrences of names of days (*Monday* ... *Sunday*).

3.6. Special symbols

The following regular expression:

```
(the + a) <MOT> is
```

allows you to find all of the sequences made up of the word "the" or "a", followed by any form, followed by the form "is".

INTEX special symbols are written between angle brackets "<" and ">". Do not leave any blank space between the angles and respect the case (upper-case for the symbol <MOT>). If you apply correctly the former expression to the text, you will obtain the concordance below.

Note the importance of the angles; the following regular expression:

```
(the + a) MOT is
```

represents the two sequences “the MOT is” and “a MOT is”. There isn't much chance of you finding that sequence in this text…

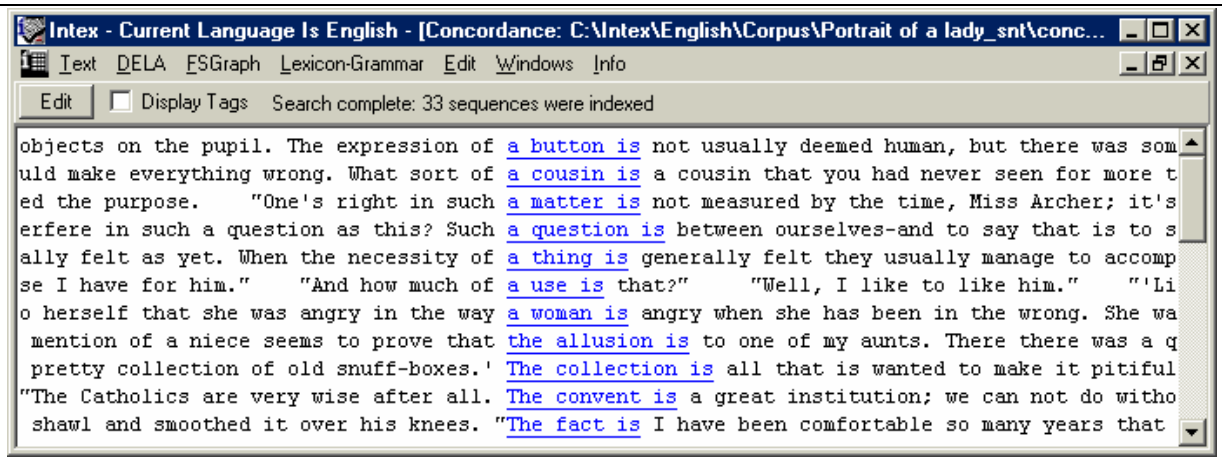


Figure 12. Search for a sequence with a special symbol

IMPORTANT: <MOT> is a special symbol. In INTEX, all **special symbols** are written inside "angles".

Following is a list of categorical symbols as well as their meaning:

Special Symbol	Meaning
<MOT>	*simple form (Sequence of letters)*
<MIN>	*simple form in lower-case (sequence of lower-case letters)*
<MAJ>	*simple form in upper-case (Sequence of upper-case letters)*
<PRE>	*Sequence of one upper-case letter followed by lower-case letters*
<NB>	*Sequence of digits*
<PNC>	*delimiter (one character)*
<^>	*Beginning of a text unit* [1]
<$>	*End of a text unit*
<L>	*letter* [2]
<U>	*upper-case letter*
<W>	*lower-case letter*

Following are a few expressions that contain special symbols:

We are searching for all the sentences that start with a form whose initial is in upper-case, then follows with a form in lower-case, and then a colon:

```
<^> <PRE> <MIN> :
```

(Apply this to the text "Portrait of a lady ", you should get one match).

Now we want to locate the forms in upper-case that appear in the beginning of sentences, or after a comma, and are followed by the form “said”:

```
(<^> + ,) <PRE> said
```

(there are 29 occurrences in the text). Now we will locate all sequences of two consecutive forms written in upper-case letters:

```
<MAJ> <MAJ>
```

[1]. Linguistic units are either the line or paragraph (by default), or the sentence, the article or anything useful (if the text has been pre-processed).

[2]. The symbols <**L**>, <**U**> and <**W**> are used only at the morphological level. Letters are the characters that are explicitely listed in the **Alphabet** file of the current language.

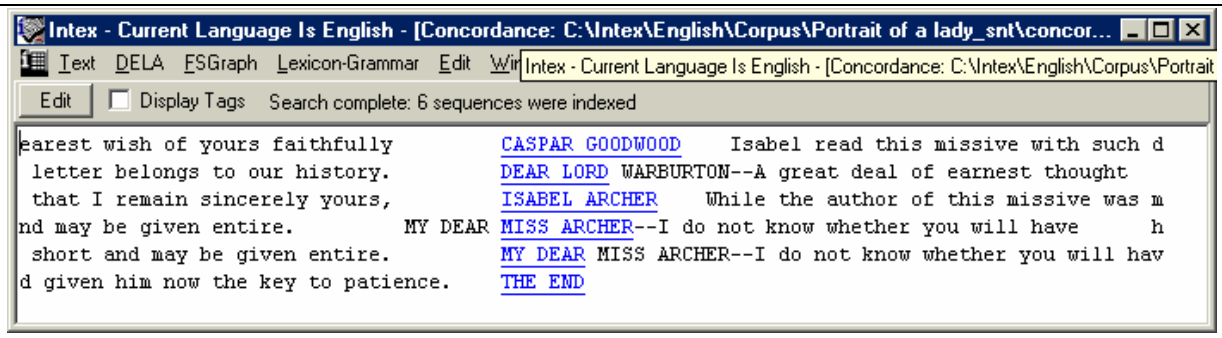


Figure 13. Search for a sequence of two upper-case forms

Now locate the forms that appear between "at" or “in” and “of” (there should be 142 matches):

```
(at + in) <MOT> of
```

3.7. Special characters

The Blank

Remember that INTEX considers any sequence of spaces, tabulation characters, and line change (codes "NEW LINE and "CAR RET") characters as one "**blank**". When entering a regular expression, blanks are usually unrelevant and are therefore optional.

Generally, one does not search for spaces:

- In morphology, locating is limited to the simple word, the space is therefore never present;
- In syntax, the space is always implicit; the expression <MOT><MOT>, for example, recognizes any sequence of two simple forms (that are naturally always separated by a space).

The following expression, for example:

```
<NB>  ,
```

recognizes any sequence of consecutive digits that are directly followed by a comma, but **also** those that are followed by a blank (in INTEX terms, i.e., any sequence of spaces, line changes, or tab characters). Both of the following sequences are recognized by the previous expression:

```
1985,
1734          ,
```

However, it is sometimes necessary to specify the "mandatory", or "forbidden" spaces; in which case, we can use the quotation mark to isolate the space. The following is a valid regular expression:

```
<NB> " "  ,
```

that recognizes only digit sequences that are followed by at least one space and a comma. Note that between the space and the comma, there might be extra spaces.

However, the following expression recognizes only the digit sequences that are directly followed by a comma (without a space):

```
<NB> # ,
```

How do we enter the query “a sequence of digits followed by exactly one space, and then a comma”? the following regular expression can be used:

```
<NB> " " # ,
```

the sharp character (“#”) matches if and only if there is no blank at the current position in the text. Note that the following regular expression will never recognize anything:

```
<NB> # " "
```

because if right after the sequence of digits, there is no blank, then the “ “ will never match.

Quotation marks and the backslash "\"

Quotation marks are used in INTEX to protect any sequence of characters that would otherwise have a particular meaning in the writing of a regular expression (or, as we will later discover, of a tag in a graph). For example, if we want to locate in a text all the single forms in parentheses, we would enter the expression:

```
"(" <MOT> ")"
```

Similarly, if we want to locate uses of the character “+” between numbers:

```
<NB>  "+"  <NB>
```

If we want to protect a single character, we can also use the protection character (\ ) as a prefix. The following expression is identical to the one just above:

```
<NB>  \+  <NB>
```

Note that if one wants to locate the characters “\” and the quotation marks in a text, one has to protect them, by adding an extra “\” before, i.e. \\

Quotation marks are not useful in the following case: the expression in the following line is simpler and equivalent:

```
"1234"  "&"  "VXII"  "."
    1234  &  XVII .
```

Quotation marks are used to perform **exact** matches. Note that the following two expressions are not equivalent:

```
"is"  "A:B"
  is  A:B
```

"is" in the first expression only recognizes the lower-case form “is”, and not “IS” nor “Is”; "A:B" does not recognize the variations with a space such as “A : B”.

The sharp character "#"

The sharp character is used to forbid the use of a space. For example, when locating decimal numbers with a comma (and to avoid confusion with the use of the comma as punctuation), one could use the following expression:

```
<NB> # , # <NB>
```

3.8. The empty string "<E>"

The <E> special symbol represents the empty string, in other words the neutral element of the concatenation operation. It is generally used to note an optional or elided element. For example, to represent the two variables:

```
a credit card + a card
```

One can use the following, more compact version:

```
a (credit + <E>) card
```

Similarly, if one wants to locate the utterances for the form "is" followed within a context of two words, by "the", “this” or “that”, one can use both of the following expression:

```
is ((the+this+that) + <MOT> (the+this+that) + <MOT> <MOT>
(the+this+that))
```

But the next expression is in general more compact and legible:

```
is (<E> + <MOT> + <MOT> <MOT>) (the+this+that)
```

3.9. The Kleene operator "*"

The Kleene operator is used to indicate any number of utterances. For example, if one is locating the matches for the form "is" followed by any number of forms, followed by the form "the", the following expression would be used:

```
is <MOT>* the
```

Note that the number of forms is unlimited and includes zero: the previous expression is equivalent to the following infinite expression:

```
is (<E> + <MOT> + <MOT> + <MOT><MOT> + …) the
```

The following expression:

```
the very* big house
```

recognizes an unlimited number of sequences:

```
the big house, the very big house, the very very big
house, the very very very big house,...
```

When using the Kleene operator to specify an insertion of unlimited length, be careful not to forget potential delimiters. For example, to recognize the sequences made up of the form "is", then of a possible insertion, then of the form "by", you should enter the expression:

```
is (<MOT> + <PNC>)* by
```

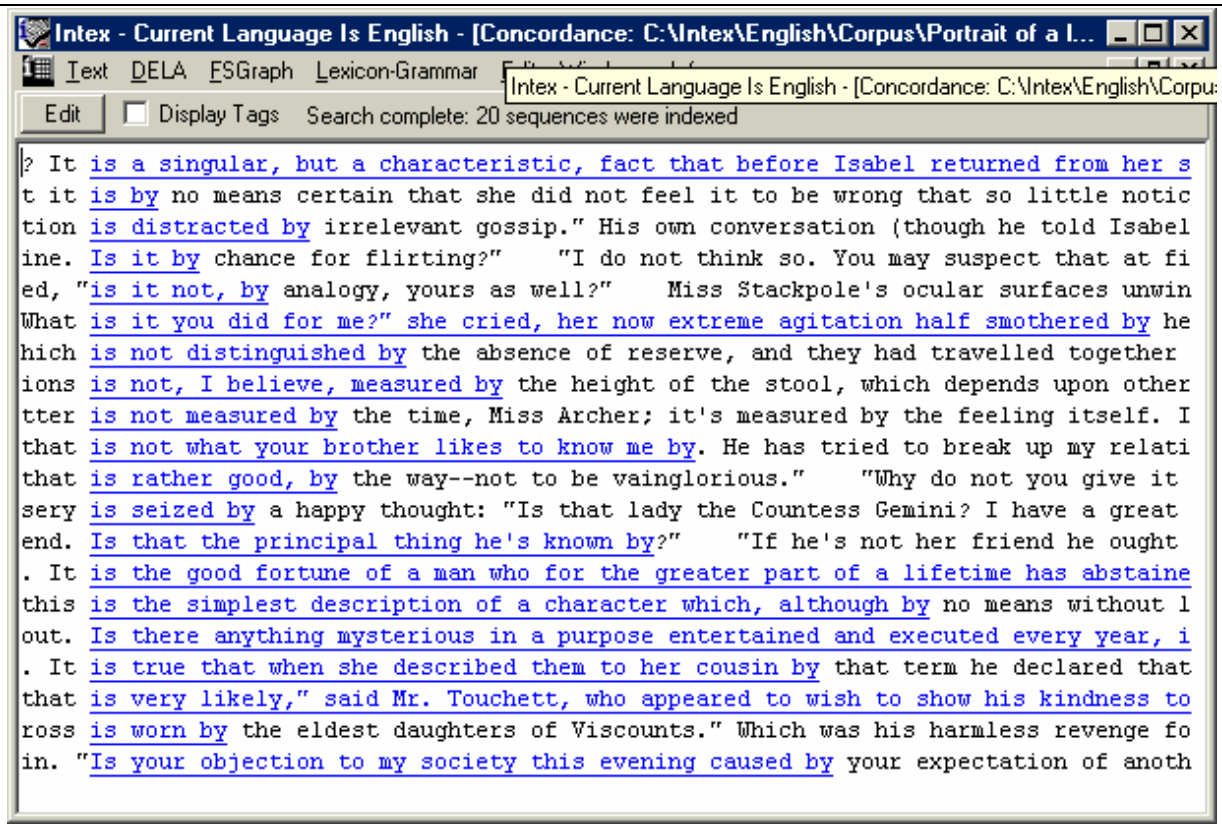


Figure 14. Arbitrary sequences in a pattern

(Note that you can change the length of the left and right contexts in the concordance).

In morphology, the following expression recognizes all of the words that have "de" as a prefix and "ation" as a suffix, particularly the forms *destructuration*, *desorganization*, *destabilizations*, etc.

```
<^> d e (s+<E>) <L>* (z + <E>) (a t i o n) (s+<E>) <$>
```

Summary:

You have learned to write regular expressions:

-- the blank (concatenation operator) allows you to build sequences of words;
-- the “+” (disjunction operator) allows you to select alternate sequences;
-- the "*" (Kleene operator) is used to mark non-limited repetitions.
-- the <E> symbol (the empty string) is the neutral element for the concatenation.

The Kleene operator takes priority over concatenation, which takes priority over disjunction ("or" operator). Parentheses can be used to change the order of priority.

Chapter 4. USING LEXICAL RESOURCES

4.1. Indexing all inflected forms of a word

Previously, we located the conjugated forms of the verb "be" thanks to the following expression:

```
am + are + is + was + were
```

We could also add the following form to the expression:

```
be + been + being
```

While this would be a perfectly valid regular expression, that would certainly be very tedious.

For each language, INTEX accesses a dictionary DELAF (further described later) in which each simple form of that language is associated to its lemma. Consulting this dictionary offers the possibility to refer to a group of inflectional forms by mentioning their lemma. The following expression, in which we refer to the lemma "be", therefore represents all of the forms in which we are interested.

```
<be>
```

Enter this regular expression in INTEX. Type it in exactly as it is above: do not confuse the angles "<" and ">" with the brackets "[" and "]"; do not insert any spaces; make sure that you type the lemma in lower-case. Apply this expression without any limitations to the text "La femme de trente ans". INTEX should find 9,010 utterances. Verify in the concordance that all the forms were located.

Re-launch the search but without typing in the angles. This time, INTEX only locates the utterances for the form "be" (1,366 occurrences).

In a regular expression, when a form is written as is (e.g. *be*), INTEX locates the utterances of the form itself. On the other hand, when the form is set between angles, this form represents a canonical form (generally a lemma); INTEX then locates all of the forms that are associated with the lemma in the dictionaries that were applied to the text.

4.2. Indexing a category

The DELAF dictionary associates a lemma as well as a morpho-syntactic category to each simple form. We may then refer to this category in regular expressions. For example, to locate all of the sequences containing any form associated with the lemma "be", followed by a preposition, then a noun, enter the following expression:

```
<be> <PREP> <N>
```

Launch the search; INTEX should show 330 sequences. Construct the resulting concordance.

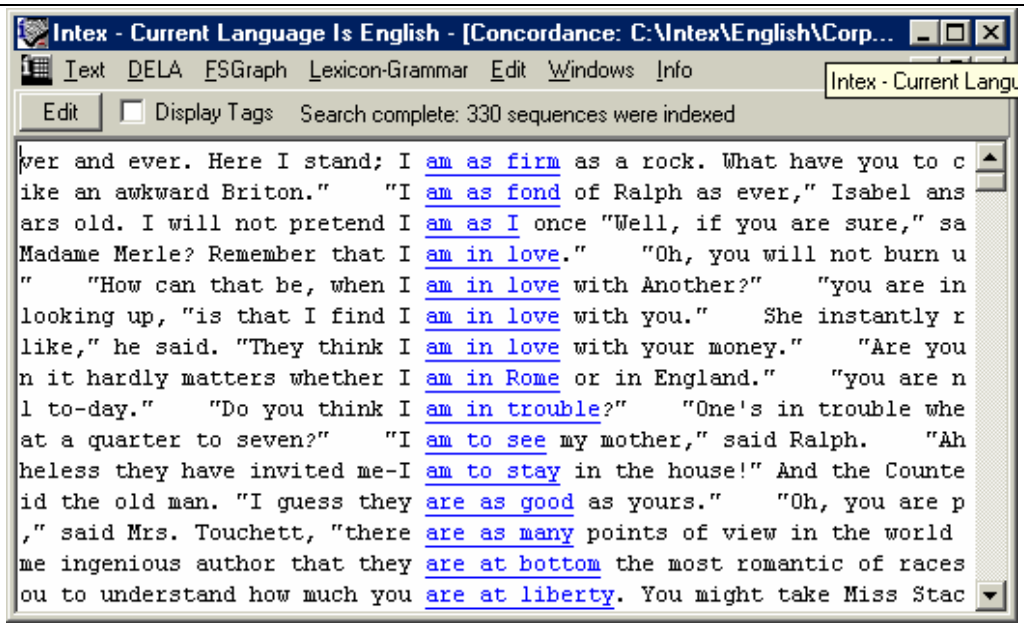


Figure 15. Use lexical information in regular expressions

In the English DELAF dictionary, **PREP** stands for **Preposition**, and **N** denotes **Noun**. INTEX will locate all of the sequences made up of a form associated in the DELAF dictionary with the lemma "be", followed by a form associated in the DELAF with the “PREP” category code, followed by a form associated in the DELAF with the “N” category code.

The following symbols are references for the codes found in the English DELAF:

Code	Meaning	Examples
A	*Adjective*	*artistic, blue*
ADV	*Adverb*	*suddenly, slowly*
CONJC	*Coordination conjunction*	*and*
CONJS	*Subordination conjunction*	*if, however*
DET	*Determiner*	*this, the, my*
INT	*Interjection*	*ouch, damn*
N	*Noun (substantive)*	*apple, tree*
PREP	*Preposition*	*of, from*
PRO	*Pronoun*	*me, you*
V	*Verb*	*eat, sleep*
X	*non-autonomous constituants of compounds*	*extenso, priori*

These codes are not set by INTEX. Rather, they are a part of the dictionaries accessed by INTEX. In other words, INTEX does not know what the symbol "ADV" means: in order to recognize the special symbol <ADV>, INTEX consults the system dictionaries and verifies if the word is therein associated with the code **ADV**.

Important: Users may add their own morph-syntactic codes to the system, either in new, personal dictionaries or by modifying the system's dictionaries. The new codes must always be written in upper-case. They are immediately usable in any regular expression or grammar.

Before adding new codes to the system, you should verify that they do not conflict with codes used in other dictionaries. For example, do not enter a list of occupations with the code <PRO> if you plan to use the DELAF dictionary, because in this dictionary, this code is already used to mark the pronouns.

Conversely, if you add a list of terms that have the function of a substantive, it is preferable to code them “N” rather than, say, “SUBS”, so that the grammars you write may refer to all nouns, including the ones that are described in other dictionaries.

We will now locate the sequences of the form "be", followed by an optional adverb, a preposition, then the determiner “the”. Reactivate the **Locate** window and enter the following expression:

```
<be> (<ADV> + <E>) <PREP> the
```

Click on Start, INTEX finds the corresponding sequences; construct their concordance.

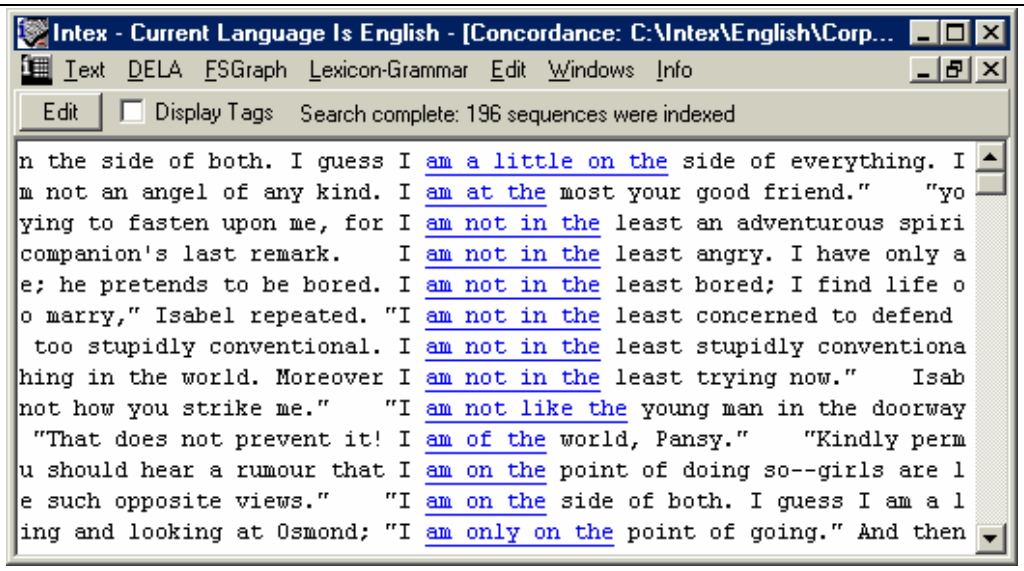


Figure 16. Another regular expression

Note that INTEX recognizes not only simple words, but compound words as well. For instance, the first sequence “am a little on the” was recognized because “a little” is described as an adverb in the dictionary for compounds.

4.3. Combining lexical information in symbols

DELAF type dictionaries contain at least a lemma and a morpho-syntactic code associated with each entry (form). They may contain other types of information, and all of the lexical information available in these system dictionaries may be used. For example, here is an entry from the English DELAF dictionary:

```
abuses,abuse.V:P3s
```

This entry states that the form “abuses” is linked to the canonical form “abuse” (in this case, its infinitive from); this is a verb (V), conjugated in Present (P), 3rd person singular (3s).

There are two types of information that can be encoded in a DELAF dictionary: syntactic-semantic information, and inflectional information.

Syntactic and semantic information

This information is represented by codes introduced by the character "+"; the syntactic or semantic codes can be any character string that contain no space, line change, "+" nor ":". For example:

```
viruses,viruse.N+Conc+Medic:p
```

could represent the fact that the noun "virus" acts as a concrete noun (as opposed to an abstract noun such as "process"), and is used in a medical semantic domain.

Warning: the "+" character in the dictionary has nothing to do with the "or" operator in regular expressions.

One can insert these codes in lexical symbols, to the right of a lemma or of a category. For example, `<fly+t>` denotes transitive uses of the verb *to fly*, and <V+4> represents all verbs that are described in the syntactic table 4 of the lexicon-grammar[3].

One can combine these codes as much as much as needed. For example <N+Hum+z1> represents human (+Hum) nouns belonging to the basic vocabulary. (+z1). Syntactic and semantic codes are not ordered: for instance, the previous symbol is equivalent to <N+z1+Hum>.

[3] In the lexicon-grammar, table 4 describes all the verbs that have a sentence as a subject and a human noun as a direct object, e.g.: *the fact that it rains (amuses+annoys+upsets) John*.

On peut combiner ces codes à volonté ; par exemple, <N+Hum+z1> représente les noms humains appartenant au vocabulaire de base. Les codes ne sont pas ordonnés : le symbole précédent est équivalent à <N+z1+Hum>.

Warning: Codes are case sensitive. For example, the codes "+Hum", "+hum", "+HUM" would represent three different codes to INTEX, and the symbol <N+Hum> would not match a lexical entry associated with the code "+HUM" or "+hum".

Inflexional Information

This information is represented by code sequences introduced by the character ":"; each inflectional code is represented by a single character which cannot be a blank, a "+" or a ":".

Warning: here too, case is relevant. In the English DELAF, for example, the inflectional code "P" represents the present tense while the code "p" represents the plural.

A lexical entry may be associated to no inflectional code (for example, prepositions), to one single inflectional code (the infinitive forms of verbs, for example), or to several (for example, a verb conjugated in the present tense, in the third person, plural).

If there is an inflectional ambiguity, the form is associated with more than one sequence of inflectional codes; each sequence of codes being introduced by the "**:**" character. For example, the French form "*aidions*" is associated with the following DELAF entry:

```
aidions,aider.V:I1p:S1p
```

"aidions" is the form of the verb "aider" (*to help*) conjugated either in the imperfect tense (I), first person (1) plural (p), or in the present subjunctive (S), first person (1) plural (p).

In French, many verbal forms are five times ambiguous. For instance, the verb form "aide" would also be associated with the following lexical entry:

```
aide,aider.V:P1s:P3s:S1s:S3s:Y2s
```

In the French DELAF, "P" stands for "Present"; "S" for "Subjunctive" and "Y" for "Imperative". Possible inflectional code combinations naturally depend on each language. The choice of the codes is open and can be modified by users, as long as there is no ambiguity.

Here are the inflectional codes used in the English DELAF dictionary:

Code	Signification
s	*Singular*
p	*Plural*
1, 2, 3	*1st, 2nd, 3rd person*
P	*Present tense*
I	*Preterit*
K	*Past participle*
G	*Gerundive*
W	*Infinitive*

Both in symbols and in dictionaries, inflectional codes are not ordered. For example <V:P3s> and <V:3Ps> recognize the same lexical entries. INTEX allows partial queries, for example, <be:P> represents all of the forms of the verb *to be* conjugated in the Present tense, and <be:3s> matches both forms "is" and "was".

4.4. Negation

INTEX processes three levels of negation in regular expressions:

- one can request all the sequences that were *not* recognized by a regular expression or a graph; for instance, one can extract all sentences that do not contain any verb by entering the regular expression <V>, and then selecting the option "**Extract all non-matching units**" in the window "**Display Indexed Sequences**";

- one can match all word forms that are not associated with some lexical information, by prefixing the symbol with the character "!". For instance, <!V> matches all the word forms that are not verbal forms; <!have> matches all the word forms that are not associated with the lemma "to have"; <!N:Hum:p> matches all the word forms that are not plural human nouns;

- one can prefix all the syntactic and semantic features of a lexical entry with the character "-" instead of the character "+"; in that case, only word forms that are *not* associated with the feature will match; for instance, <N-Hum> matches non-human nouns, <V-t> matches intransitive verbs.

Warning: negations often appear to produce obscure, inexpected results in INTEX, because of the huge lexical ambiguity both in dictionaries and in texts. For instance, in the following untagged text:

```
I left his address on the table
```

the query <!V> would match all word forms, including “left”, because this form is also associated with the lexical entry left = Adjective; <!N> also matches all the forms, including “address” and “table”, because both forms are associated with lexical entries that are not nouns (the verbs “to address” and “to table”).

As a consequence, I strongly suggest to limit the use of the negation in regular expressions to apply to texts that are largely disambiguated. As a matter of fact, all these problems disappear if one works with the following partially tagged text:

```
I {left,leave.V} his {address,.N:s} on the {table,.N:s}
```

Here, the expressions <!V> and <!N> would produce the expected results.

4.5. Tags

INTEX “.snt” text files contain four types of tokens: simple word forms, digits, delimliters and tags. We already saw that one can locate word forms (by using or not lexical information); one can also locate tags. For instance, in a regular expression, the following symbol:

```
{was,be.V:J3s}
```

matches the same exact tag everywhere it occurs in the text. The symbol:

```
{be}
```

matches all the tags of the text whose lemma is “be”; the symbol:

```
{V}
```

matches all the tags that include the code “V” (verbs), etc. Generally, symbols between curly brackets “{“ and “}” are identical to the ones between angles “<” and “>”: the only difference being that symbols between curly brackets only match tags in texts, i.e. disambiguated forms, whereas symbols between angles match both ambiguous words (both simple and compound forms) and disambiguated ones (i.e. tags).

Finally, note that if a text has been totally disambiguated (i.e. when working with a fully tagged text), angles and curly brackets are equivalent.

4.6. Exercises

(1) Extract the passive sentences from the novel “The portrait of a lady”.

One way to do that is to look for all the conjugated forms of the verb “to be”, followed by a past participle and the preposition “by”, in order to find sequences such as “... were all broken by...”.

(2) The word form “*like*” is ambiguous because it is either a verb (e.g. “I like her”) or a preposition (e.g. “like a rainbow”). Build the concordance of this form in the text; from this concordance, design two regular expressions that would disambiguate the form, i.e. one regular expression to recognize the verbal form, and one to recognize the preposition.

Start by studying the unambiguous minimal contexts in which this form is unambiguous, for instance “I like”, “(should + will) like”, “to like”, etc.

(3) Extract from the text all the sentences that express futur.

First extract sequences that contain “will” or “shall” followed by an infinitive verb; then extend the request to find constructs such as “I am going to V” and “I don’t work tomorrow”.

Summary:
Symbols in regular expressions represent:
-- word forms characterized by their case; e.g. <MIN> matches all lowercase word forms;
-- word forms associated with a lemma; e.g. <be:P> matches all forms of the verb “to be” conjugated in the Present;
-- word forms associated with a morpho-syntactic category; e.g. <N+Hum:p> matches human nouns in plural.

Chapter 5. THE GRAPH EDITOR

Until now, we have used regular expressions in order to describe and retrieve simple morpho-syntactic patterns in texts. Despite their easy use and power, regular expressions are not well suited for more ambitious linguistic projects, because they do not scale very well: as the complexity of phenomena grow, the number of embedded parentheses rises, and the expression as a whole quickly become unreadable. This is when we use INTEX graphs.

5.1. Create a graph

In INTEX, grammars of all sorts are represented by graphs. A graph is a set of **nodes**, some of them being possibly **connected**, in which one distinguishes one **initial node**, and one **terminal node**. In order to describe sequences of letters (at the morphological level) or sequences of words (at the syntactic level), one must “spell” these sequences by following a **path** (i.e. a sequence of connections) that starts at the inital node of the graph, and ends at its terminal node.

Select in the menu **FsGraph** the command **New**. A window like the following figure is displayed; this graph contains already two nodes: the initial node is represented by an horizontal “T”, and the terminal node is represented by a square in a circle. You can move these nodes by dragging them: move the initial node to the left of the graph, and the terminal node to the right, just like in the following figure:

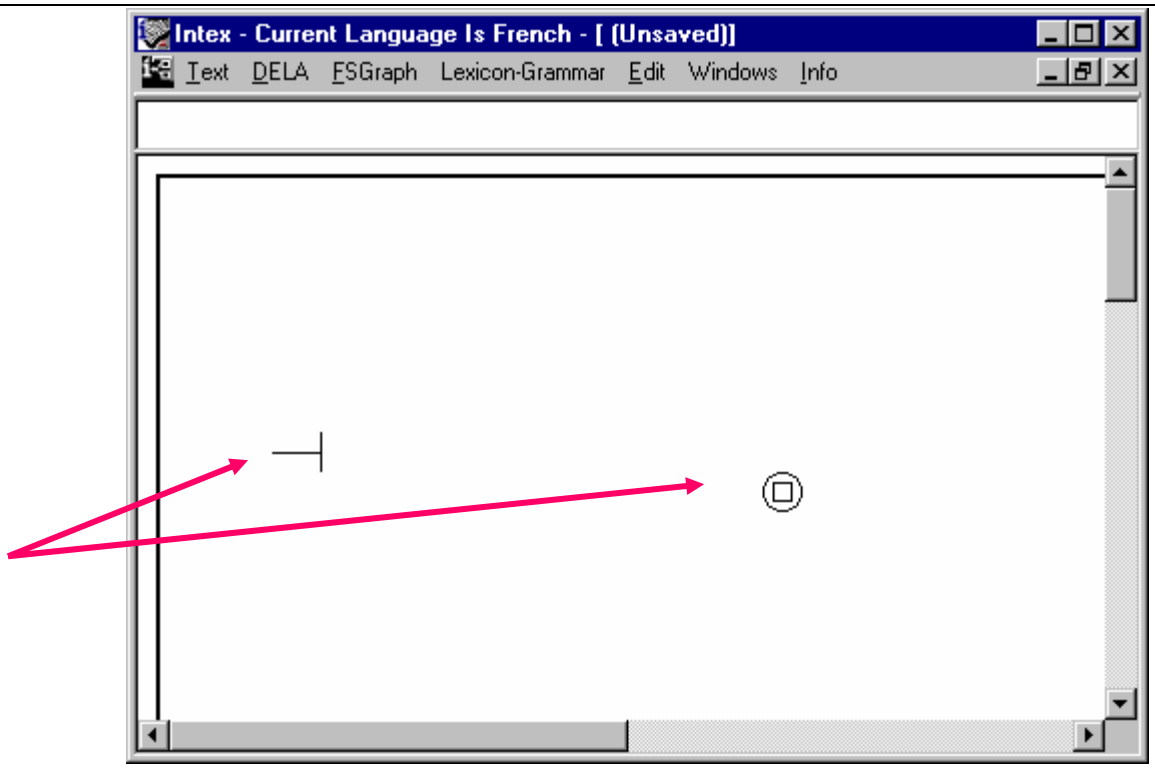


Figure 17. An empty graph contains already an initial node and a terminal node

A few operations

In order to **create a node** somewhere in the window, position the cursor where you want the node to be created (anywhere but on another node), and then **Ctrl-Click** (i.e. hit one of the **Ctrl key** on the keyboard, keep the key down, then click with the left button of the mouse, then release the **Ctrl key**) ;

When a node has just been created, it is selected (it should be displayed in blue, by default). Enter some text (this will be the label of the node), then validate by hitting the **Enter** key;

In order to **select a node**, click it. In order to **unselect a node**, click anywhere on the window (but not on a node);

In order to delete a node, select it (click it), then erase its label, then validate with the **Enter** key (a node with no label is useless, therefore it is deleted);

In order to **connect two nodes**, select the source one (click it), then the target one. In order to **unconnect two nodes**, perform the same operation, as if you wanted to connect them again: click the source one, then the target one.

Warning: if you double-click a node, INTEX understands that you connect the node to itself; therefore it creates a loop on this node. To cancel this operation, just double-click again this node: that will delete the connection.

Now is your turn!

Create the following graph:

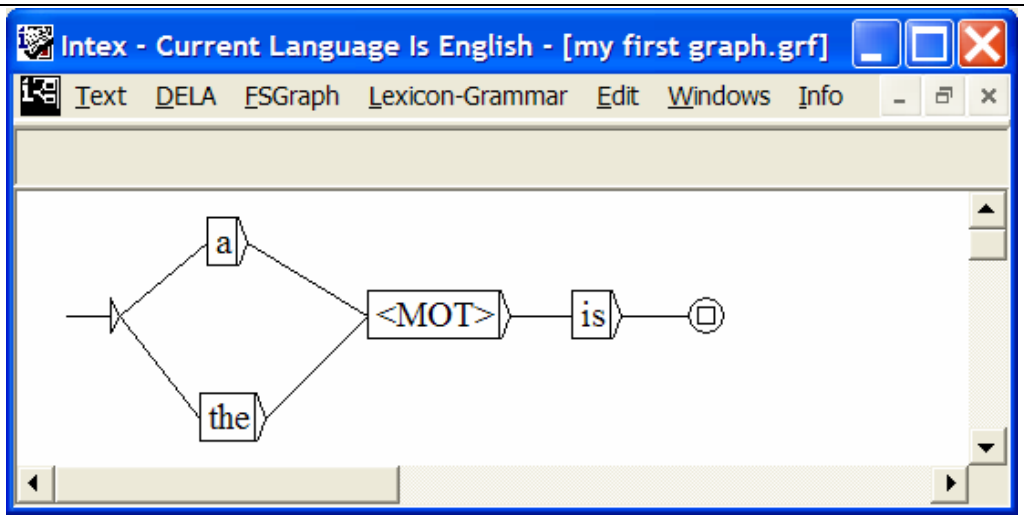


Figure 18. Graph that recogizes "a" or "the", followed by any word form, followed by "is"

In order to create this graph from an empty one, create a new node (i.e. Ctrl-Click somewhere in the graph window), enter the label "a", (i.e. check that the right node is selected, then type in the text "a" without the quotes), then validate with the Enter key. Create a second node that contains the label "the", then a third with the label "<MOT>" (use the angle brackets "lower than" and "greater than"), then a fourth in which you enter the label "is". Then, connect the nodes in the following order: connect the intial node to the "a" node by selecting (i.e. clicking) the intial node, then selecting the "a" node. Then connect the initial node to the "the" node (click again the initial node, then click the "the" node); then connect the "a" node to the "<MOT>" node, the "the" node to the "<MOT>" node, then the "<MOT>" node to the "is" node, then the "is" node to the terminal node.

You might make mistakes, such as:

-- create an extra, unwanted node; in that case, just selected the unwanted node, then delete its label, then validate by hitting the Enter key (this destroys the unwanted node);

-- create extra, unwanted (loops or reverse) connections; in that case, select the source node of the unwanted connection, then select its target node (this destroys the connection).

This graph reconizes all the sequences that start with "a" or "the", followed by any word form ("<MOT>" is a special symbol that stands for any word form), and ends with "is". For instance: "a cat is", or "the house is".

When the graph is finished, select in the menu **FsGraph** the command **Save**:

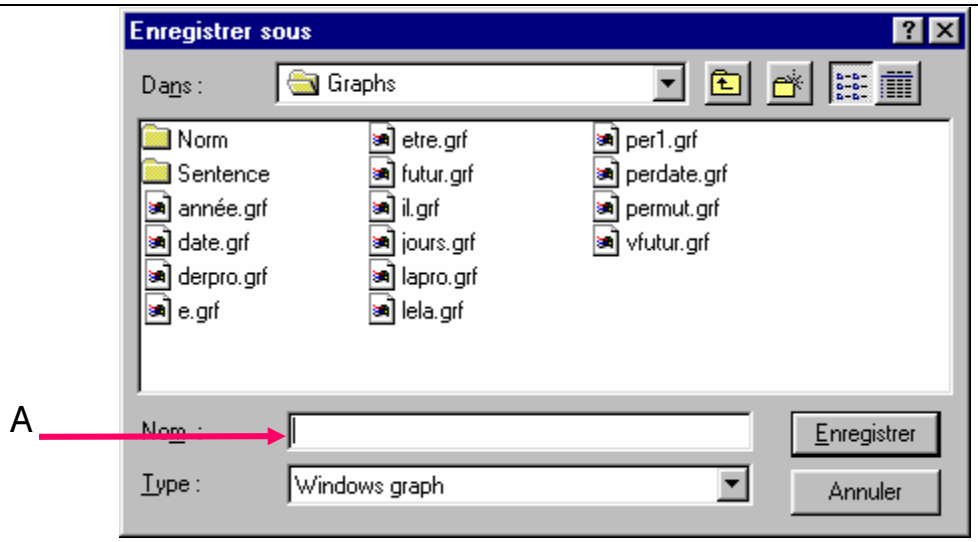


Figure 19. Nominalisation et enregistrement du graphe

Enter a filename (A) (for instance, "**my first graph**"), then validate by hitting the **Enter** key.

5.2. Apply a graph to a text

As soon as a graph is saved, one can immediately apply it to any text. If you have not already done that, load the text "A portrait of a lady.snt", then call the Locate Panel (**Text > Locate Pattern**):

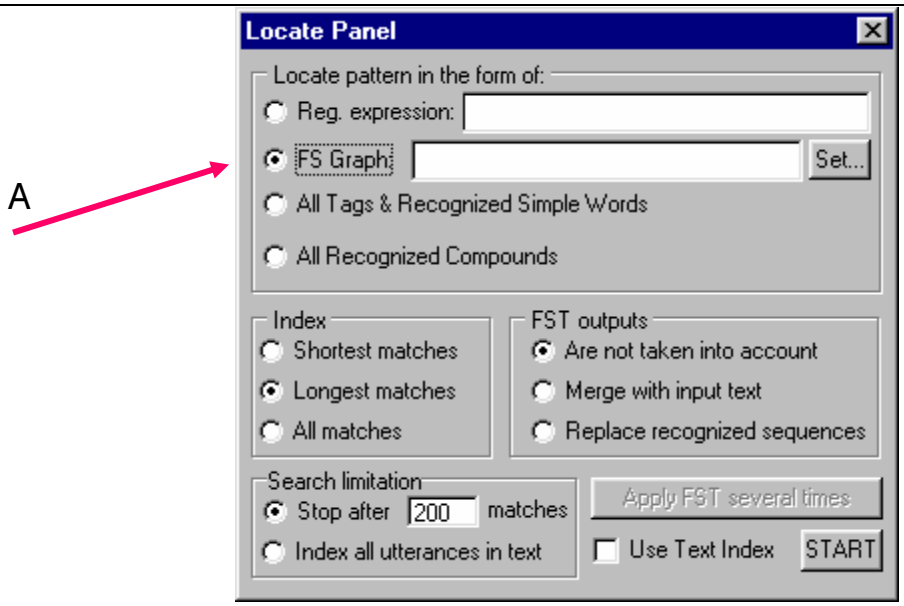


Figure 20. Applying a graph to a text

This time, instead of entering a regular expression, we are going to apply a graph. (A) Select the option **FSGraph**; a dialog box is displayed, that asks you to enter a graph name; enter it (or select it from the file browser), then validate (hit the **Enter** key, or click the **Open** button).

Finally, click the **START** button at the bottom right end of the “Locate panel”. INTEX launches the search, then displays the number of matches found; click **OK**. Build the concordance. You should get a window similar to the following figure:

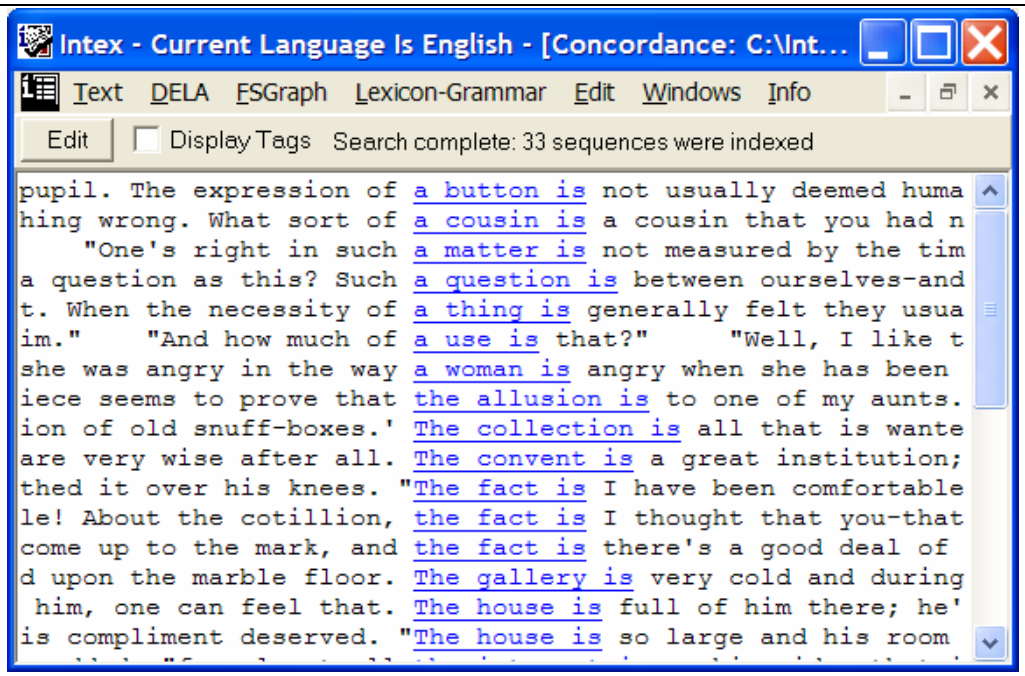


Figure 21. Concordance of the graph equivalent to the regular expression: (a+the) <MOT> is

5.3. Create a second graph

Select in the menu **FsGraph** the command **New**; a new empty graph is displayed (that already contains the initial and terminal nodes). Create three nodes with the following labels:

```
there+this, is+are+was+were, no+none+not+nothing
```

Reliez les nœuds entre eux comme dans la figure suivante.

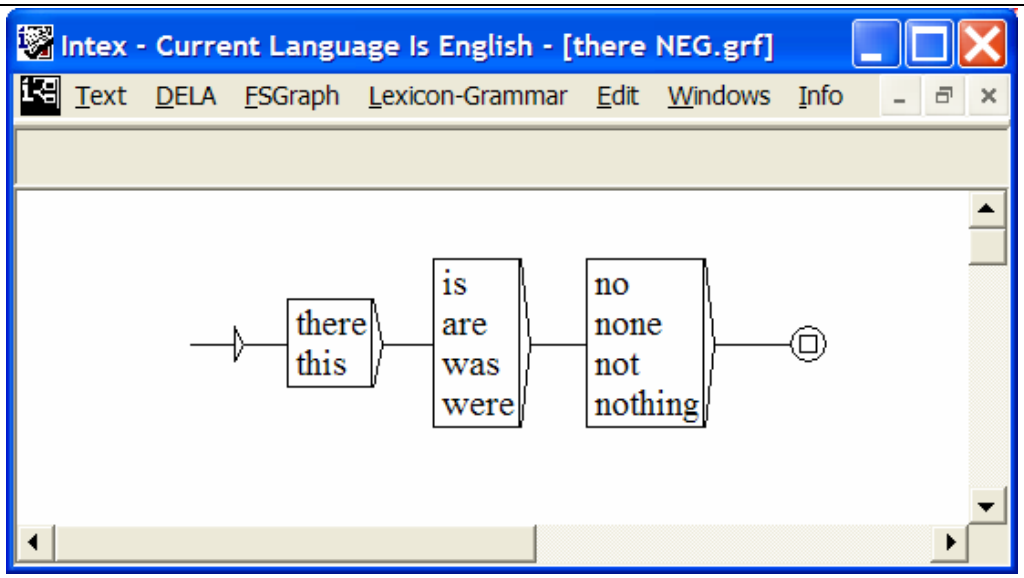


Figure 22. Another graph

Save the graph (**FSGraph>Save**), open the Locate panel (**Text > Locate Pattern…**), select the option **FS Graph**, select your graph file name, then click **START**. INTEX applies the graph to the text, and then gives the number of matching sequences; click OK, then build the corresponding concordance by clicking the button "**Build concordance**" in the window "**Display Indexed Sequences**".

Note that if lexical resources have been already applied to the text, one could have entered the symbol "`<be:3>`" (any form of "to be", conjugated at the third person), instead of the expression "is+are+was+were".

5.4. Describe a simple linguistic phenomena

Graphs are used to extract sequences of interest in texts, but also to describe various linguistic phenomena. For instance, the following French graph describes what sequences of clitics can occur between the preverbal pronoun *il* and the following verb.

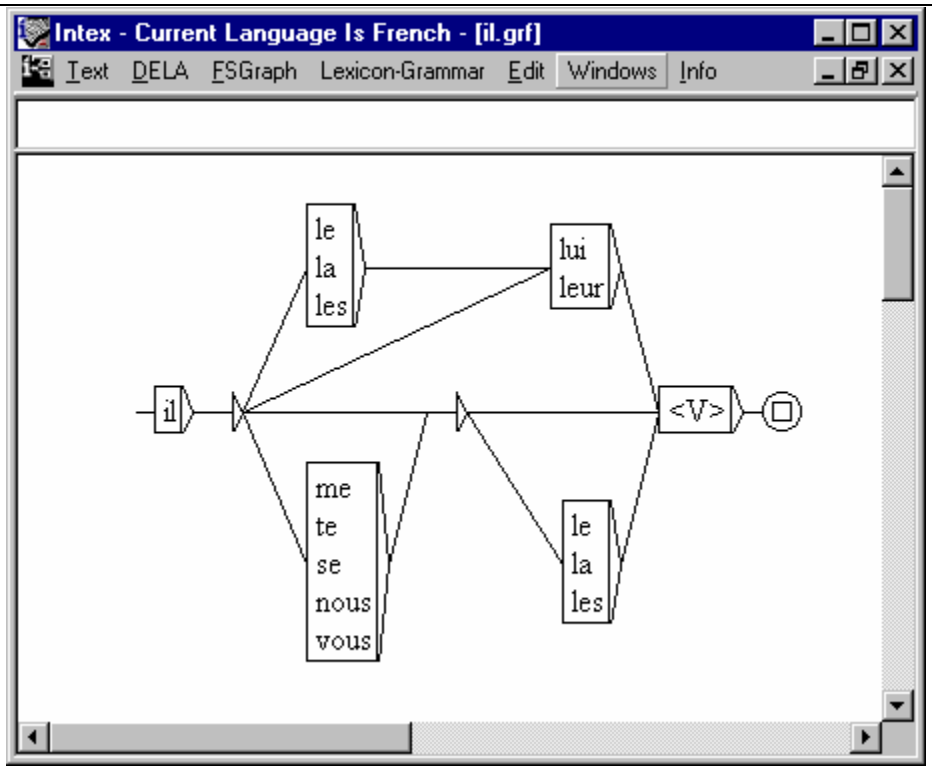


Figure 23. Une grammaire locale des pronoms préverbaux

For instance, this graph recognizes the following valid French sequences:

```
Il dort, Il le lui donne, Il leur parle, Il me la prend
```

At the same time though, the following incorrect sequences would not be recognized by the graph:

```
*Il lui le donne, *Il lui leur parle, *Il la me prend
```

Exercise: build this graph, then generalize it by adding an optional negation (e.g. “il ne lui donne (pas)”), the elided pronouns *m’*, *t’*, *s’*, and the two pronouns *en* and *y*, in order to recognize all preverbal sequences, including the following ones:

```
Il t’en donne, Il m’y verra, Il ne m’y verra (pas)
```

Then apply the graph to a French text (e.g. “La femme de trente ans.snt”) to study its coverage.

III. INTEX grammars

This section presents local grammars, and how to build libraries of graphs in order to construct a bottom-up formalization of Natural languages (chap. 6), then finite-state transducers, and how to use them to perform modifications on texts (chap. 7), then more powerful grammars, such as Context-Free grammars and Turing Machines, using Enhanced transducers and RTNs (chap. 8).

Chapter 6. LOCAL GRAMMARS

The description of certain linguistic phenomena typically requires the construction of dozens of elementary graphs such as the one we just built. At the same time, most of these elementary graphs (“local grammars”) can be re-used in different contexts, for the description of many different linguistic phenomena.

For instance, the graph **NameOfDay** that recognizes names of days (*Monday* ... *Sunday*) can be used to describe formal dates, e.g.:

NameOfDay , **NameOfMonth** (1st+2nd+3rd+4th+...+31st)
=: Monday, June 5th

to describe “informal” date complements, such as:

(<E>+last+next) **NameOfDay** (<E>+early+late) in the (morning + afternoon)
=: last Monday early in the afternoon

as well as in some specific complements:

Open **NameOfDay - NameOfDay** (1+2+...+11) (<E>+AM) (1+2+...+11) (<E>+PM)
=: Open Wednesday-Saturday 10AM-5PM

Each of these complements can be described by one graph, that will in turn be re-used in grammars that describe more and more complex phrases, up to the sentence level. INTEX takes this idea further, and allows different teams of users to cooperatively build re-usable libraries of graphs, that can be shared on one workstation, one local server, or on the INTERNET.

Following are a few examples of linguistic phenomena that are naturally described by a series of graphs.

6.1. Numeric determiners

We want to identify French numeric determiners written out in text form. (e.g. *deux mille trois cents*). We begin by constructing the two graphs "Dnum 2-99" & "Dnum 100-999" :

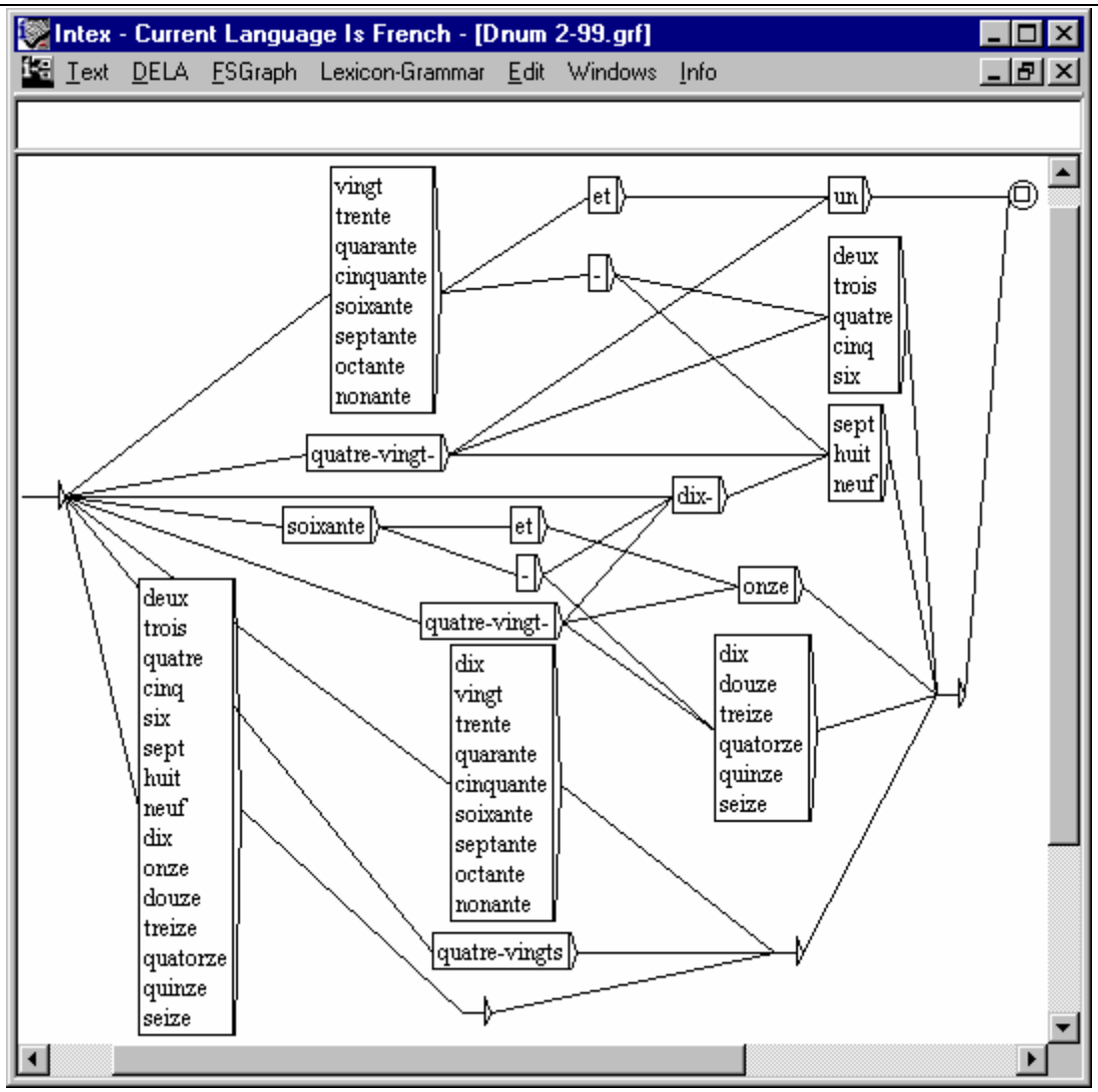


Figure 24. Numeric determiners from 2 to 99

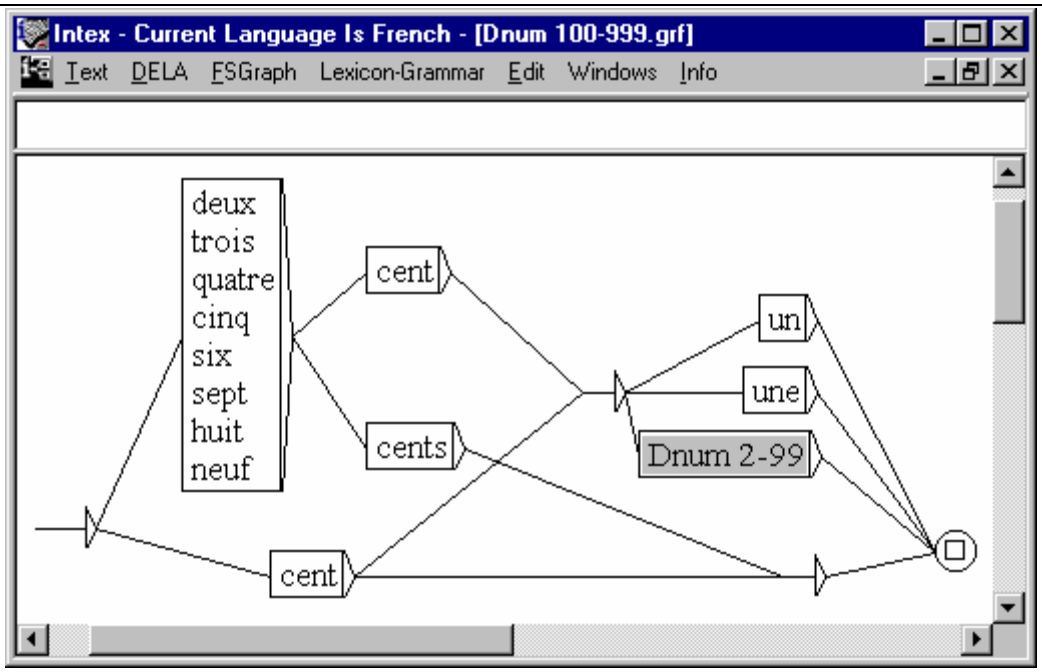


Figure 25. Numeric determiners from 100 to 999

Notice the grey node labeled "Dnum 2-99" in the graph "Dnum 100-999" : it refers to the graph of the same name. To refer to a graph, prefix its name with the symbol ":"; In this case we've entered the label ":Dnum 2-99".

To verify whether or not the imbedded graph really exists, **Alt-Click** (press the **Alt** key, and click simultaneously) on its reference: the imbedded graph should appear.

Exercise : Continue the description of numeric determinants by constructing the graph "Dnum 1000-999999" which will represent the numbers 1,000 to 999,999 , then the graph "Dnum million-milliard" which will represent the numbers containing the words million or billion *(milliard)*. Finish by constructing the graph "Dnum" which will recognize all numeric determiners written out as words :

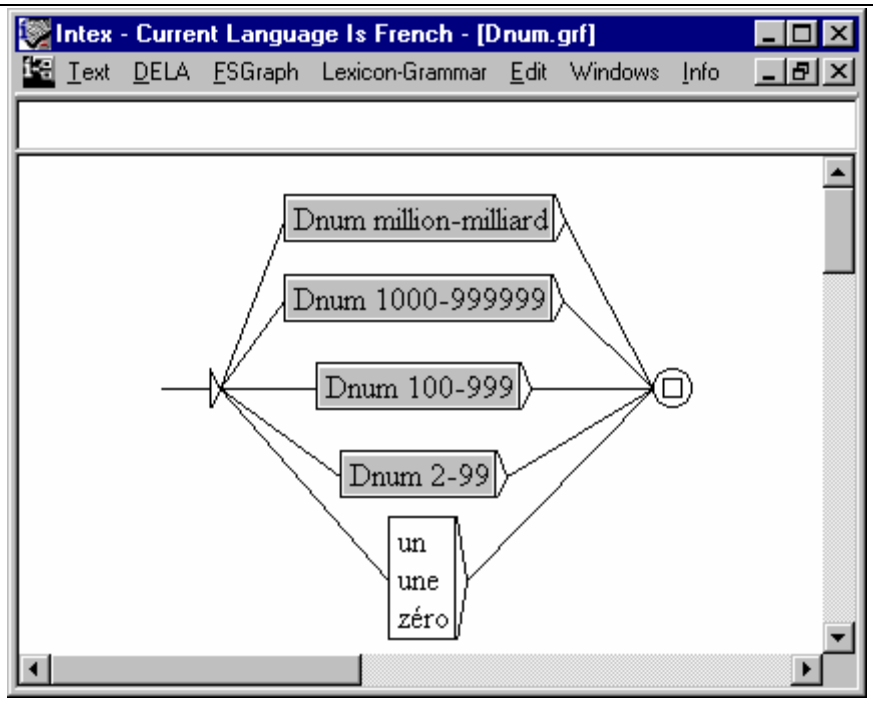


Figure 26. Graph ***Dnum***

Some pitfalls to avoid *(particular to French numeric determinants)* :

- Don’t forget that "*un*" & "*une*" are required in "*un million, mille et une"* but not in * *un mille* ;
- We write "*quatre-vingt mille"* (and not * *quatre-vingts mille*), and "*deux cent mille"* (as opposed to * *deux cents mille*). You will therefore have to have two versions of the graphs "Dnum 2-99" & "Dnum 100-999", depending on whether they are applied at the end of the determiner (ex. *quatre-vingts tables*) or in the middle (*quatre-vingt mille*).

6.2. Determiners

More generally, if we want to represent the determiners, we could draw the following general **DET** grammar:

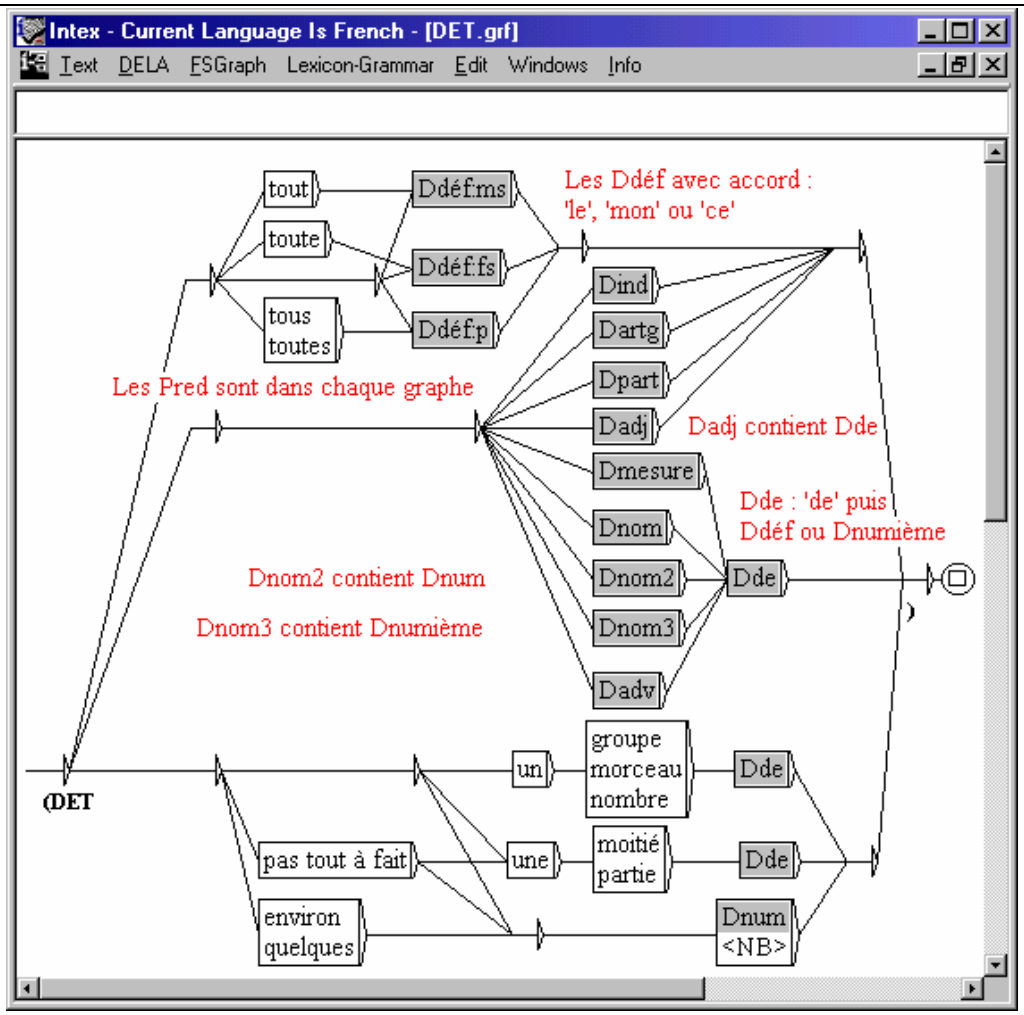


Figure 27. French determiners

This graph is based on that of A. Marchand 1999, who formalized the description given by M. Gross 1988. The DET graph contains references to other graphs : the three Ddéf graph represent the definite determiners (e.g. "*cette*"), Dind the indefinite determiners (*une*), Dpart partitive determiners (*de la*), Dadj adjectival determiners (*certaines*), Dadv adverbial determiners (*beaucoup*), Dnom the nominal determiners (*une foule*), Dnum the numeric determiners (*cinquantre-trois*). The graph DET itself is used in the syntactic module of INTEX, in the section of graphs used to identify nominal groups.

6.3. Roman numerals

Here is a simple example of orthographic description, applied to the roman numerals:

```
I, II, III, … CXXXIX …
```

It is out of the question to create a graph containing all of the roman numerals (here we will arbitrarily stop at 2999) ; Rather, we will create three graphs to represent the units, the tens, and the hundreds as well as a fourth graph to bring together the first three. Here first is the graph representing ones:

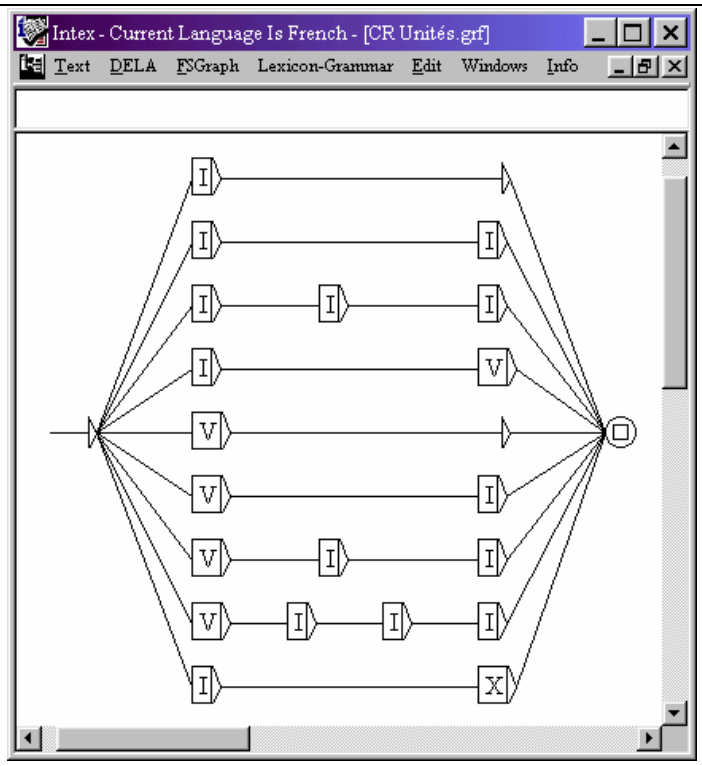


Figure 28. Roman numerals, the ones

The graph representing the "tens" is constructed in a similar fashion. :

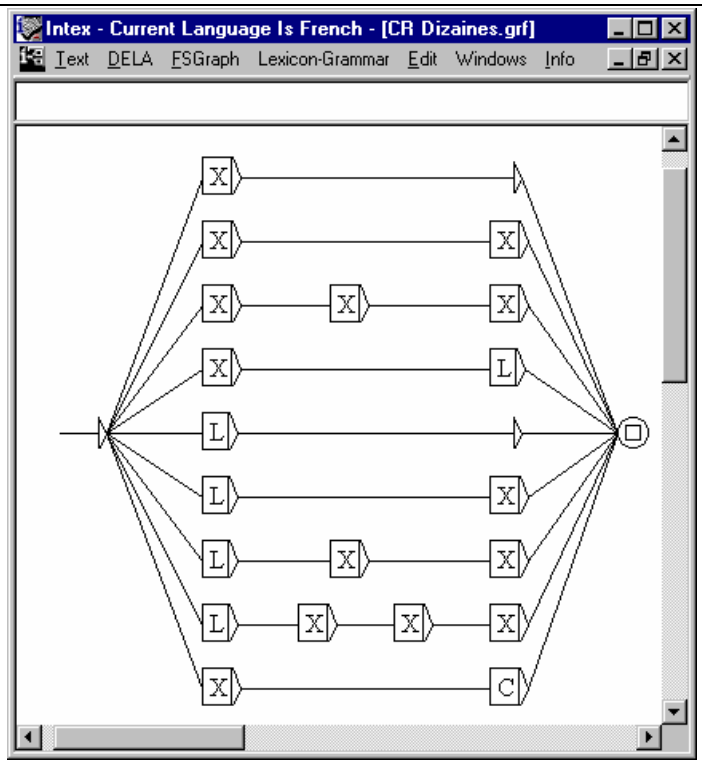


Figure 29. Roman numerals, the tens

Notice the remarkable similarity between this graph and the former one. (constructed by replacing "I" by "X" and "V" by "L") ; the graph describing the hundreds is therefore:

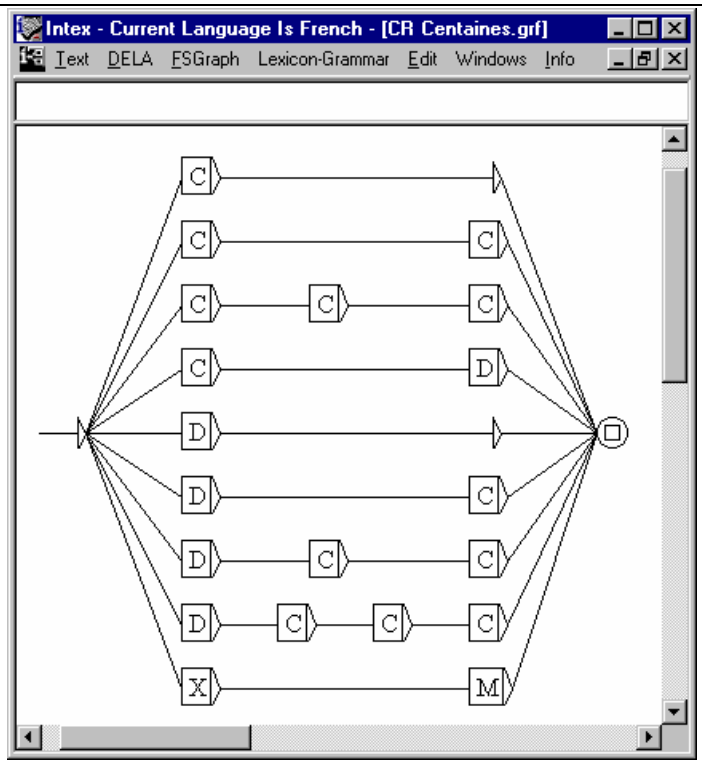


Figure 30. Roman numerals, the hundreds

The graph for roman numerals will therefore be:

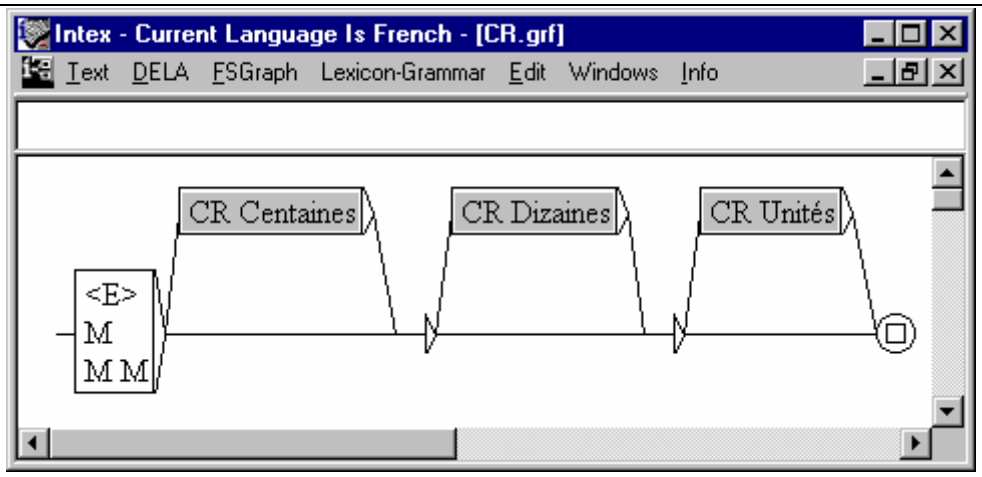


Figure 31. Roman numerals

The grey nodes in this graph refer to the imbedded graphs of the same name ; the "greying" of these nodes is obtained by prefixing the name of the node with the character ":". In other words, the label of the first node is ":CR Centaines". You can open an imbedded graph by **Alt-Clicking** the node (press **Alt**, and click simultaneously).

6.4. Resolving the references for graphs

There are several ways, within a graph A to refer to another graph B:

- We can indicate the file name and path for graph B as in the following example:

```
c:\\Mon Intex\\French\\Graphs\\Syntaxe\\Dét\\Dadv.grf
```

Note that you must double the backslash character "\" each time, since it is a special character, with a particular usage within INTEX.

- If the file name extension for B is not indicated, INTEX will look first for a windows format graph ("`.grf`" extension), then a NextStep format ("`.graph`" extension), then finally a graph compiled in transducer format ("`.fst`" extension);

- We can directly relate the file name of graph B to that of graph A : so for example, "**Dadv**" can refer to a file located in the current folder (i.e. in the same folder as the file of graphe A) ; "**Dét\\Dadv**" is

referring to the file **Dadv** found in the folder **Dét** which itself is found in the current directory, etc.

- If the name of graph B is given alone (without mention of the parent folder), and B is not located in the current file, INTEX will look for B in the special folders "Graphs\Lib" of the current language, first in the personal directory of the INTEX user, then in the application folder.

Summary

You have seen how to construct graphs and re-use them by mentioning them in other graphs; this function will permit you to construct virtual libraries of graphs, and in so doing, will cover increasingly complex phenomena from morphology to syntax.

Chapter 7. TRANSDUCERS

Up to this point, the graphs have been used for the *identification* of sequences in texts. We will now see that it is possible to use graphs for the *modification* of text..

INTEX analyses will often consist of enriching the text, whether by *replacing* sequences of simple forms with lexical entries (which amounts to replacing the *forms* by *labels*, in the text file), or by *inserting* structural indicators in the text (represented again by *labels*).

The tools used to perform such modifications in the text are called transducers, which are INTEX graphs similar to those that we've already seen, but which associate **produced** information to **recognized** sequences. Transducers are used in two modes: in the "REPLACE" mode, sequences recognized by the transducer will be replaced by generated sequences; and in "MERGE" mode, the sequences produced or generated by the transducer will be inserted into the text.

7.1. Editing a transducer

Graphically, the sequences to be recognized will be written in the *nodes* of the graph; the produced sequences will be written below the nodes. The nodal labels then will contain two elements:

```
Recognized sequence / produced sequence
```

As an example, recreate the following graph:

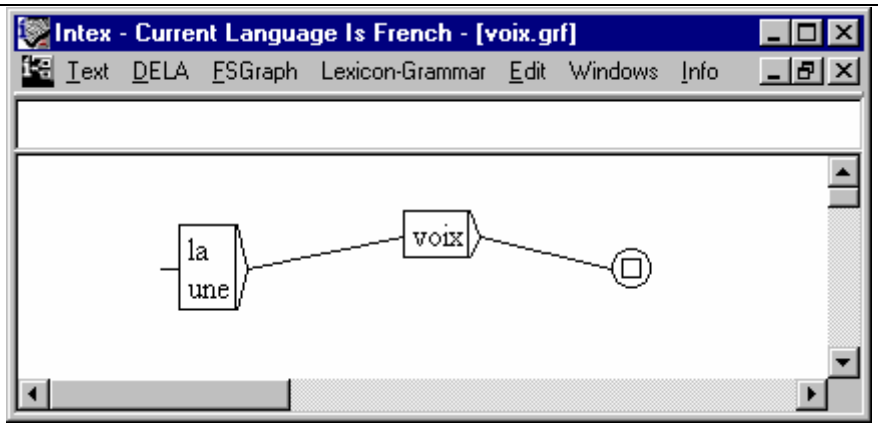


Figure 32. Graph recognizing the two sequences : la voix, *et* une voix

This graph recognizes the sequence: *la voix* and *une voix*. We will associate the mark "`[GN]`" to all sequences recognized by this graph. Select the initial node and add the character "`/`" to its label, then the sequence to be generated "`[GN]` ":

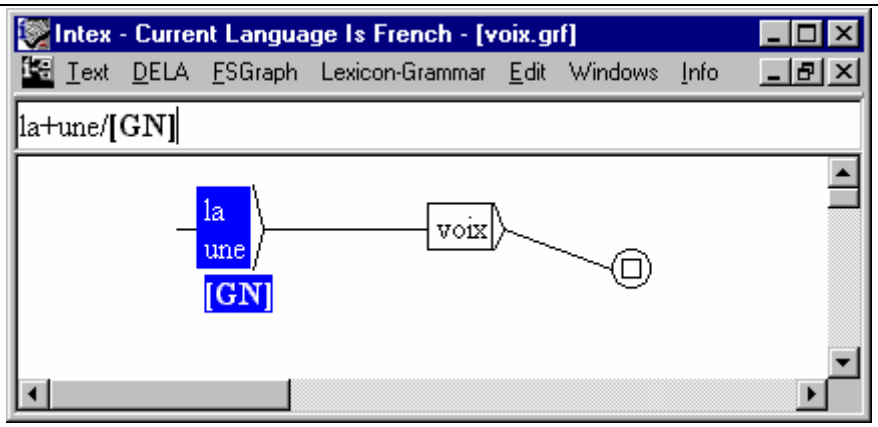


Figure 33. Production of linguistic information

Complete the entry by pressing the "**Enter**" key, then save the text (keyboard shortcut **Ctrl+S**), with the filename "voix" for example. Congratulations, you've just completed your first transducer!

7.2. Modifying text

Now we will apply this transducer to the text "La femme de trente ans.snt". Reload the text if necessary (**Text > Open**), then call up the search window (**Text > Locate Pattern**). In the field "**Locate Pattern in the form of**", select the option **FSGraph**, then enter the name of your graph :

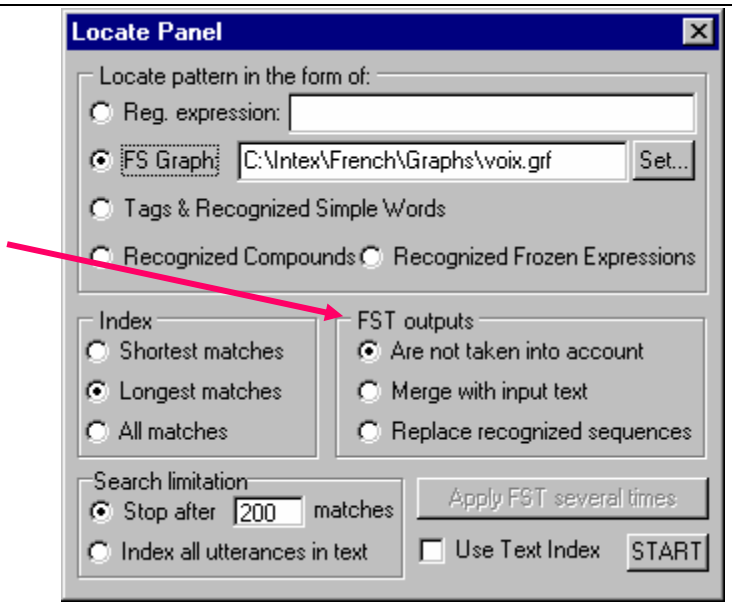


Figure 34. Applying a transducer

In the **FST outputs** zone *(what the transducer has produced …)* we can choose between three possibilities :

(1) "**Are not taken into account**": Here we simply ignore the information produced by the transducer concerning ourselves simply with the ability to identify the sequences, as before. Applying the transducer then, gives the following result:

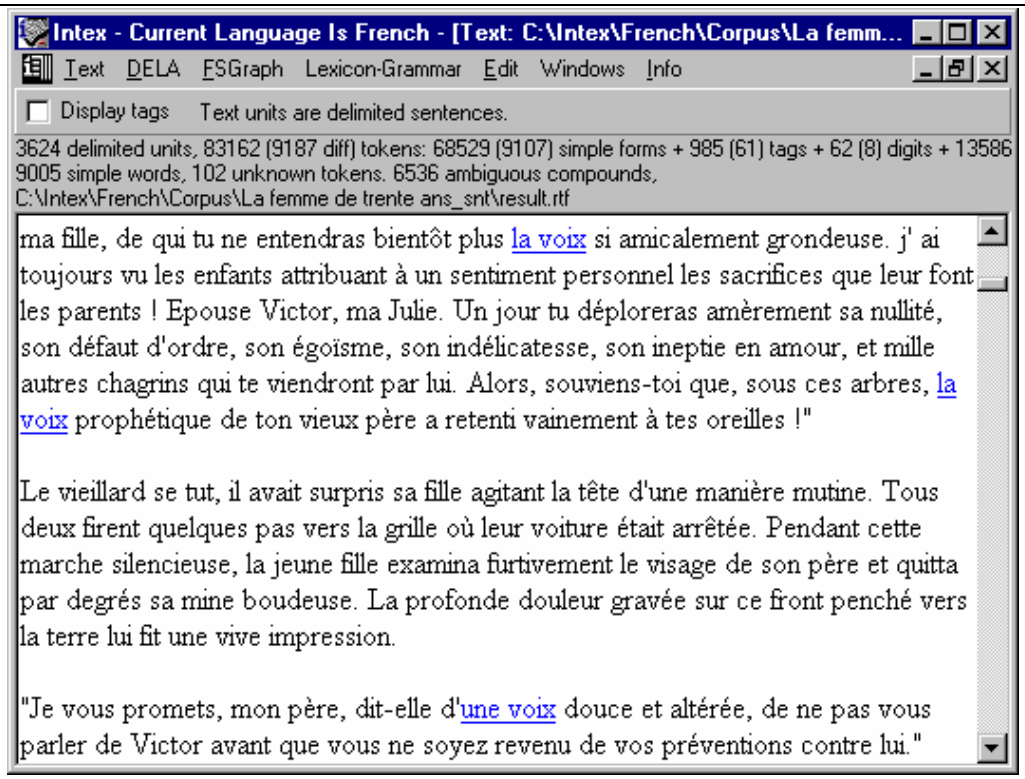


Figure 35. We seek only to identify recognized sequences

(2) "**Merge with input text":** In this case, INTEX will insert the produced sequences in the text; see the following results.

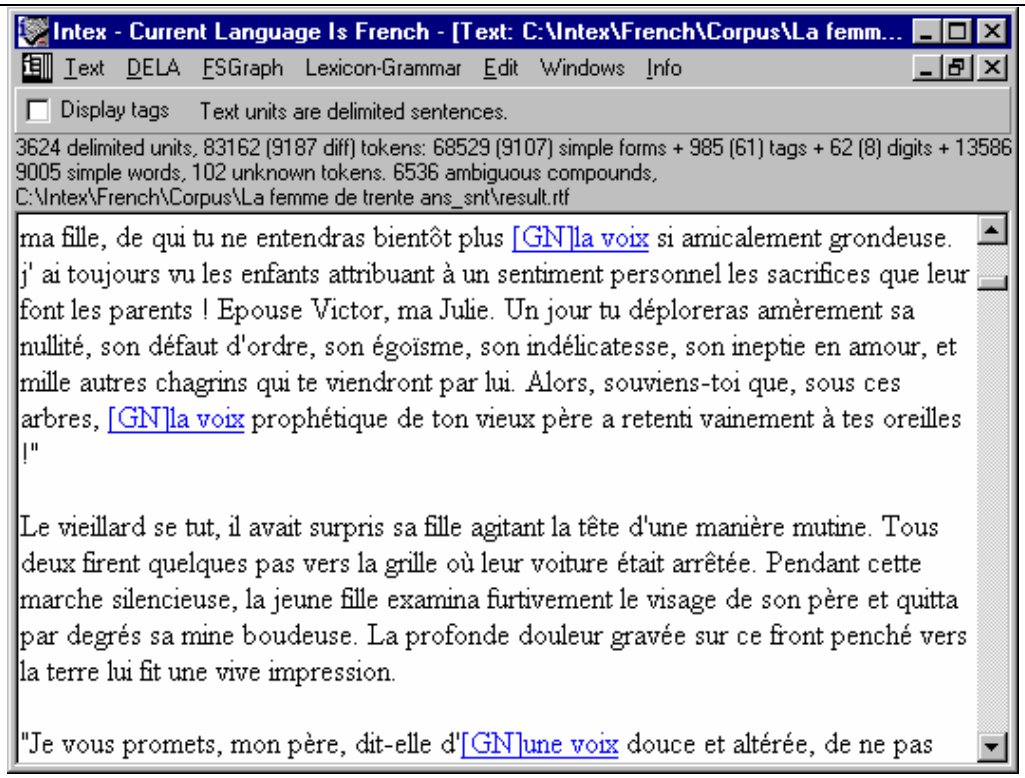


Figure 36. Insertion of a marker before each recognized sequence

Note that in when merging, the produced sequences *(written below the node in our graph)* are inserted before the recognized sequences *(in the node of our graph)*. We will see that this "node by node" synchronization allows us to perform operations, which will reduce ambiguity in a text. In these operations, each element of a given sequence will be confronted by lexical constraints.

For a better understanding of synchronization, modify the preceding graph to obtain the following:

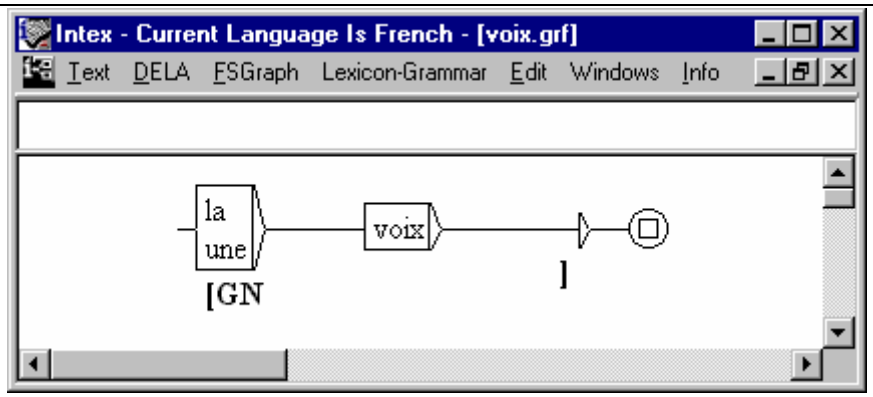


Figure 37. Variant of a transducer

In the initial node, erase the closing bracket and add a space in its place (so as not to have GN directly connected to the adjacent word in the text). Create a new, empty

node with the label "<E>/]" and connect the "voix" node to it. Connect this empty node to the terminal node and erase the link between "voix" and the terminal node. Save the graph then apply it to the same text and your will achieve the following.

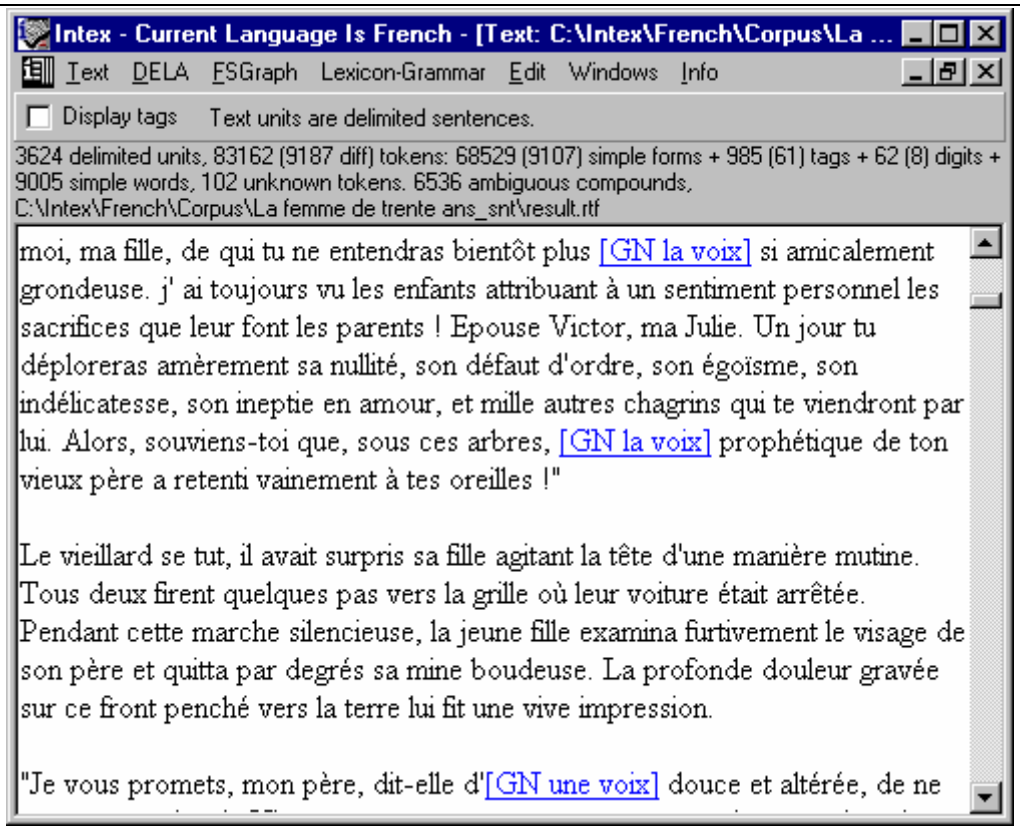


Figure 38. Writing recognized sequences between brackets

We've just added structural marks to the text. The applications of this type of operation are varied, morphological analyses (inserting morphemes to a word), indexation of graphs (inserting parentheses around complex expressions described by the graphs), syntactical analyses (where nominal groups are placed between parentheses), production of structural documents DHTML, etc.

(3) "**Replace recognized sequences**" Here, INTEX will replace the recognized sequences by the produced sequences. As an example, the two sequences "la voix" and "une voix" will be replaced by the produced sequence "[GN]":

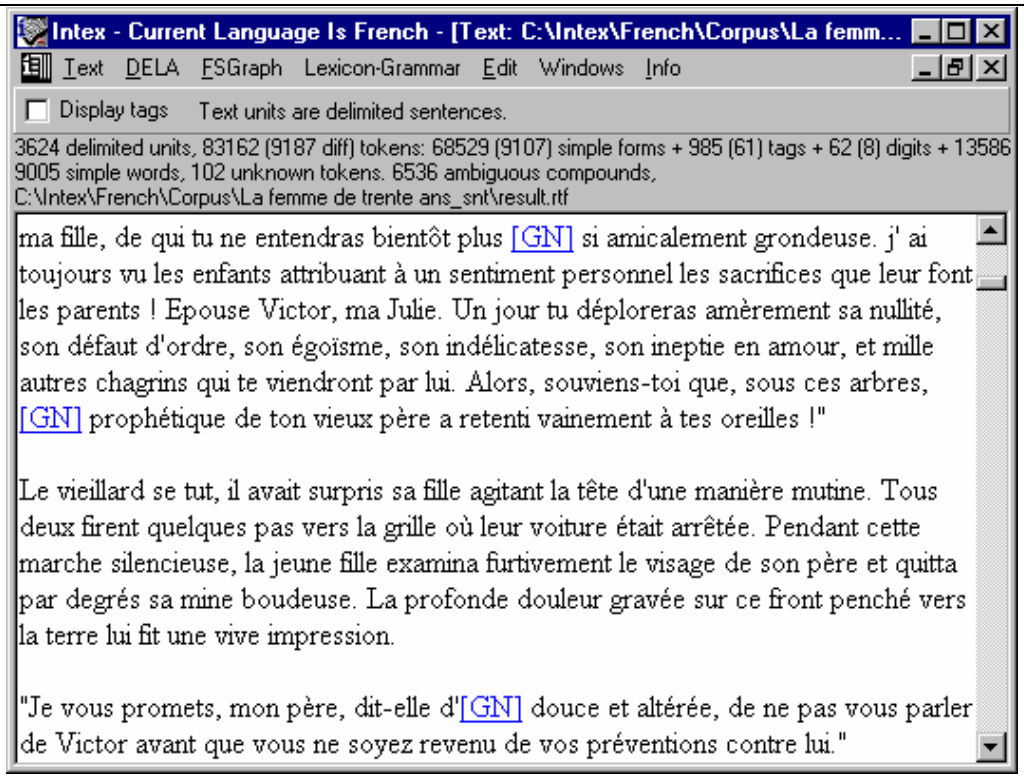


Figure 39. We replace recognized sequences by produced sequences

7.3. The five rules of transducer application

A given graph could potentially recognize an infinite number of sequences and associate to each one a different produced sequence. If we apply such a graph to texts, intending to modify them (i.e. in "REPLACE" or "MERGE" modes), the result could seem counter-intuitive. Here we intend to bring out the application of transducers using 5 simple rules.

1. <u>The graphs are always applied on a "go-forward " basis.</u>

For example, after we apply the following graph in "REPLACE" mode:

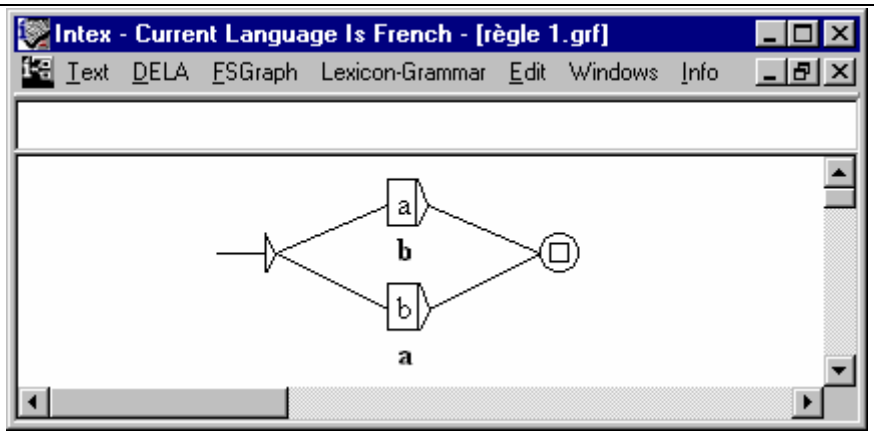


Figure 40. Two substitutions without loops

Each occurrence of "a" in the text will be replaced by an occurrence of "b", and each occurrence of "b" will be replaced by "a". For example, if the initial text were:

a b z b a b z

The resulting text would be:

b a z a b a z

This double substitution functions in parallel, because once the graph has recognized one letter (e.g. "a") and has replaced it by the produced letter ("b"), INTEX applies the graph to the remainder of the text. In other words, INTEX applies the graph progressively through the text which avoids potential blockages (as could be the case if we had to replace "a" with "b", then "b" with "a", then "a" with "b" etc.

2. Graphs are always read left to right

Consider the following graph:

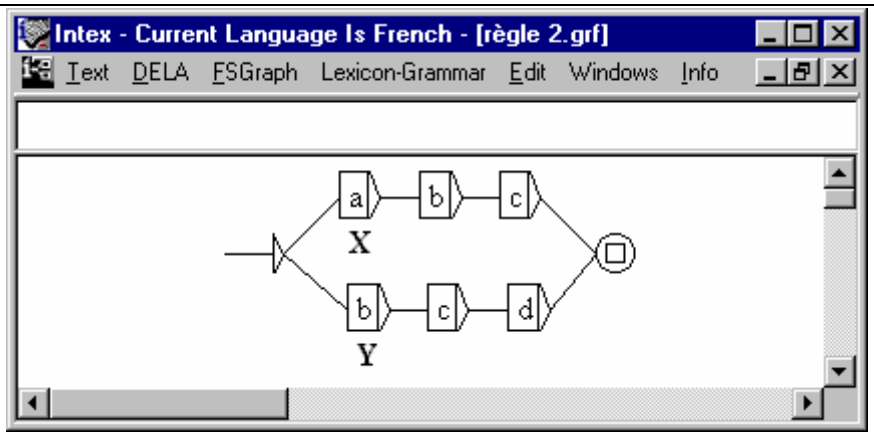


Figure 41. Two substitutions where cross-over exists

If we apply the graph to the following text:

z a b c d z

We will achieve the following result:

z X d z

In other words, the sequence "a b c" was recognized first because it is "a-initial" as opposed to the sequence "b c d". After having been identified, it is replaced by the sequence "X", then (rule #1), INTEX advances in the text. The sequence "d z" is not recognized, so it remains unchanged.

3. The Longest sequences always take priority

Consider the following graph:

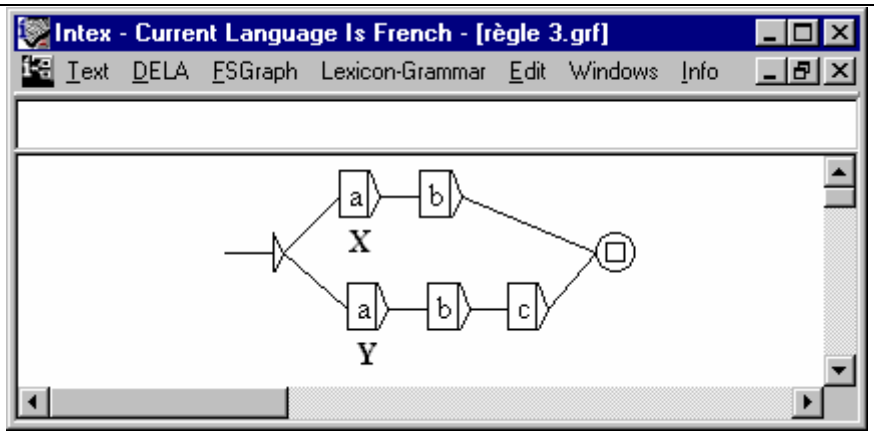


Figure 42. Two substitutions

If we apply the graph to the following text:

z a b c z a b z

We will achieve the result:

z Y z X z

Initially, the two sequences "a b" and "a b c" were recognized. INTEX replaced "a b c" with "Y" because the sequence "a b c" is longer than the sequence "a b". In other words, the insertion of "X" will only take place if INTEX recognizes an "a b" sequence that is NOT followed by "c".

4. Graphs cannot contain contradictory commands

The following transducer, applied in "REPLACE" mode asks INTEX to replace "a b" both with "X" and "Y" which makes no sense. Application of this graph would give an infinite result.

Caution: In general, INTEX cannot detect such contradictory commands. It is your responsibility to verify that your grammar, regardless of the degree of complexity, does not associate multiple entries of the same sequence with different productions.

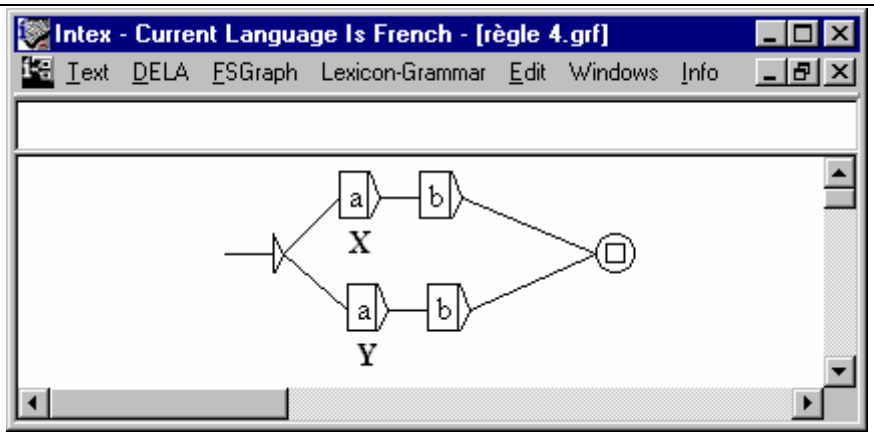


Figure 43. Two contradictory commands

The only time that this constraint becomes a non-issue is when the text, rather than being represented in linear fashion by a sequence of words, is itself represented by a graph. In this case, if a transducer furnishes multiple different productions, they will be displayed as parallel paths in the graph of the text.

5. A transducer cannot recognize an empty node:

INTEX cannot apply graphs that identify empty nodes. The follow graph, when applied in "MERGE" mode:

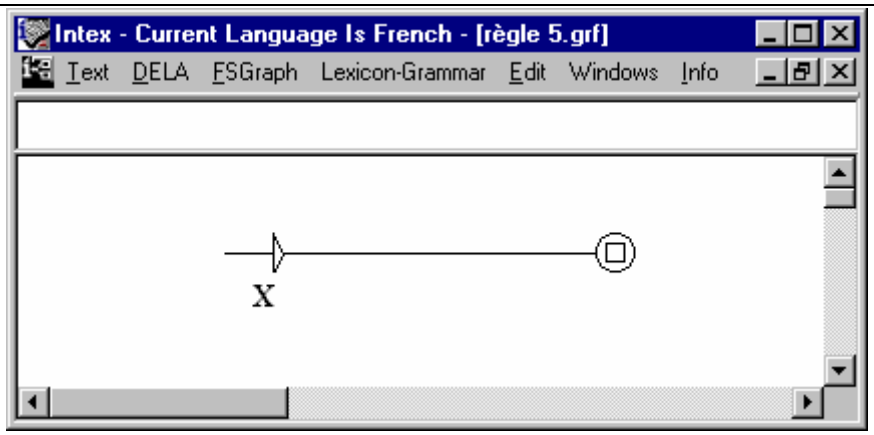


Figure 44. A graph recognizing the empty node

would require that INTEX insert an infinite number of X`s before it even read the first character of the text file! Fortunately, INTEX will recognize graphs with this flaw and will warn you.

7.4. ".grf" et ".fst" files

Graphs are stored in files with the extensions "`.grf`". Elsewhere, however, you have undoubtedly seen the file extensions "`.fst`" *(Finite State Transducers, or transducers)*; these files are compiled versions of the graphs. The differences between the two file types are:

- "`.grf`" files contain graphic information such as the position of each node, the colors used for drawing the nodes, the fonts used for text comments, node labels and labels below the nodes, etc. "`.fst`" files do not contain this information, they only contain the grammar information (labels and connections).;

- A "`.grf`" file can include one or more than one imbedded graphs which, themselves can also contain other imbedded graphs etc. The corresponding can include one or more than one imbedded graphs which, themselves can also contain other imbedded graphs etc. The corresponding "`.fst`" file represents the net information of all the hierarchically imbedded graphs.

This reduction to the net information of all hierarchical graphs consists of replacing all the references to other graphs with the information from each of those graphs. For example, if a graph **G1** contained three references to **G2**, INTEX would replace the references to **G2** with the **G2** graphs themselves.

The transducers compiled in the "`.fst`" format do not contain information that is not required when the graph is applied to the text, for example the font used, positions of the nodes within the graph, etc. Furthermore, they are optimized such that they contain no redundancy:

- A graph can contain multiple paths beginning with the same prefix. The corresponding compiled transducer only contains one occurrence of this shared prefix. The process by which redundant prefixes are eliminated is referred to as the **determinization** of the graph.

- A graph may contain multiple paths which all end with the same suffix. The corresponding compiled transducer only contains one occurrence of this shared suffix. The process by which redundant suffixes are eliminated is referred to as the **minimalization** of the graph.

For example, the following graph contains several redundancies:

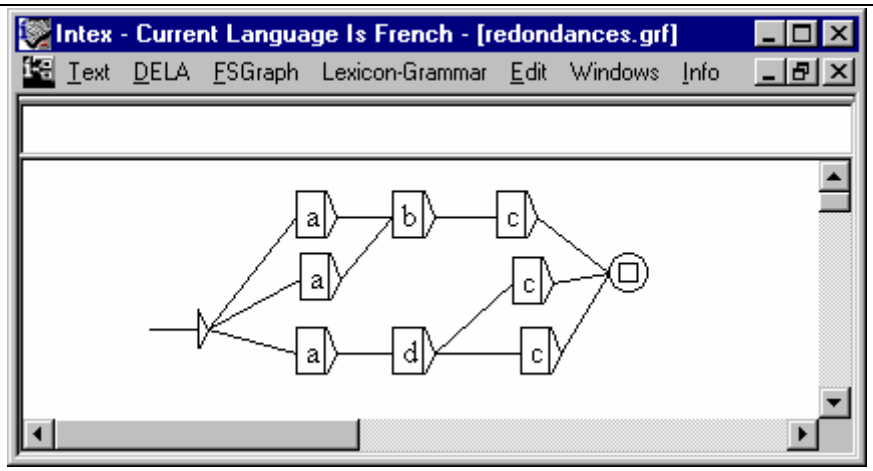


Figure 45. A graph containing redundancies

The corresponding transducer, automatically compiled would therefore be:

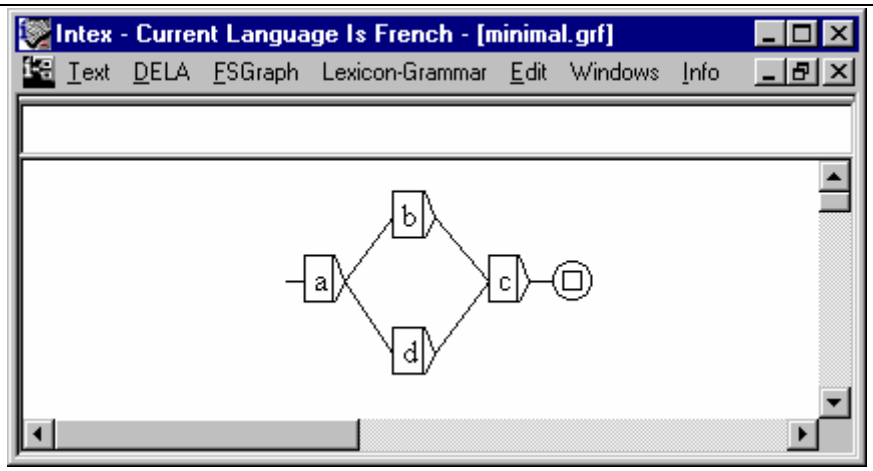


Figure 46. Corresponding minimal deterministic transducer

(The original "`.fst`" file cannot be seen ; in the preceding figure we "cheated" by constructing the image of the "`.fst`" file as a graph).

You can compile a graph by following the commands : "**FSGraph > Tools > Compile FST**" or "**FSGraph > Tools > Compile & Determinize FST**". In the "Locate" window, you can then access a transducer visually represented in the form of a "`.grf`" graph, or a compiled form "`.fst`".

When you are constructing your grammars, don't hesitate to include redundancy in the description of the linguistic phenomena, if these redundancies add clarity to the description (which is often the case). The determinization of your grammar will automatically remove these redundancies in the "`.fst`" file.
When the number of imbedded graphs remains relatively small (max. 9-10), the compiled "`.fst`" transducers are more compact than the graphs, making it sometimes quicker to apply the compiled transducer than to apply the original graph.

When the number of imbedded graphs becomes greater than ten or so, the multiple copies of the graphs which themselves contain multiple copies of the imbedded graphs, can explode the size of the corresponding "net version" transducer. We encountered a case where a grammar, made up of some 50 imbedded graphs was compiled into an "`.fst`" transducer of more than 400,000 nodes! In such a case, the compilation of the transducer can require far more time than merely applying the uncompiled graph to the text, especially if the text is relatively small or if you limit the search to a small number of occurrences.

To sum up, it will generally be more rapid to directly apply the graphs if the grammar contains several dozen graphs and if we want to quicly apply the grammar, modify it, re-apply it, etc. during the construction & debugging stage.

On the other hand, it will be more efficient to compile the grammar and apply the "`.fst`" file, if it is reasonably small or well-finalized.

Chapter 8. GOING BEYOND FINITE STATE MACHINES

8.1. *CF* Grammars

Consider the following S graph:

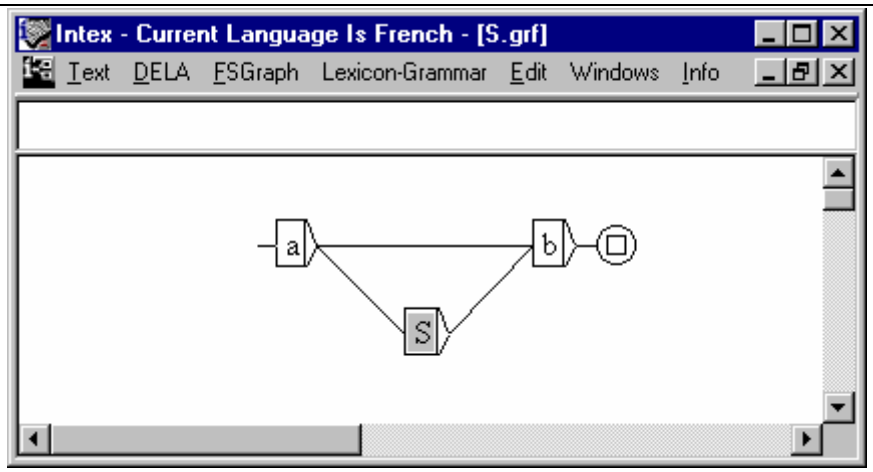


Figure 47. Example of a CF grammar

The graph is entitles S, and contains a reference to itself. This graph recognizes the following two sequences:

```
a b, a a b b
```

The sequence "a b" is recognized thanks to the upper path of the graph; the sequence "a a b b" is recognized by the following means: it recognizes the first "a", then it recognizes "a b" thanks to S as before, then "b": we then reached the terminal node, so the sequence is identified. We can, in the same manner, identify an infinite number of sequences:

a a a b b b, a a a a b b b b, etc.

More generally, any sequence composed of a given number of "a", followed by the same number of "b". This graph recognizes any number (regardless of size) of equivalent sequences, but any non-equivalent sequence will be rejected.

In this type of grammar, we cannot replace each reference to a graph by the graph itself, as has been the case up until now; these are non-finite state grammars; we call them "context-free grammars".

When a graph contains a reference to itself (as is the case of graph S), we say that it is directly recursive. Certain grammars are indirectly recursive when for example they contain a graph A which contains a reference to a graph B, which itself contains a reference to graph A.

There are three types of recursion: *head recursion*, *tail recursion* and middle recursion:

Grammars with head recursion

Consider the following recursive grammar:

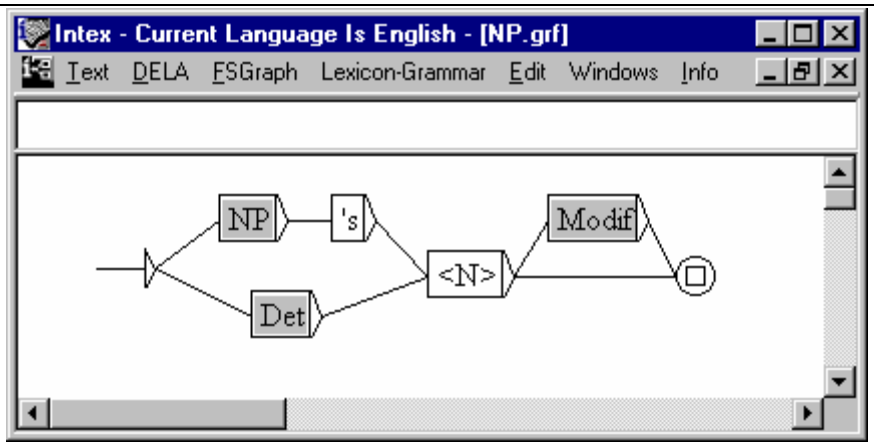


Figure 48. A Grammar with Head Recursion

This grammar represents the English nominal groups (NP = *Noun Phrase*) made up of a determiner, a noun, an optional modifier to the right, and also all the preceding nominal groups followed by "'s" (indicator of the genitive case) and a noun. It's considered head recursion because the reference to NP occurs to the left of the graph NP. It recognizes the following sequences:

```
The head of the division, the head of the division's car
   the head of the division's colleague's wife's car
```

Note that in English there is a distributional constraint NP =: N+Hum that is not taken into account here: the nominal group NP before 's must be +Hum, while the same will not necessarily be so for the entire nominal group.

Head recursions may be automatically removed from a grammar. For example, the grammar NP is equivalent to the following grammar NP2:

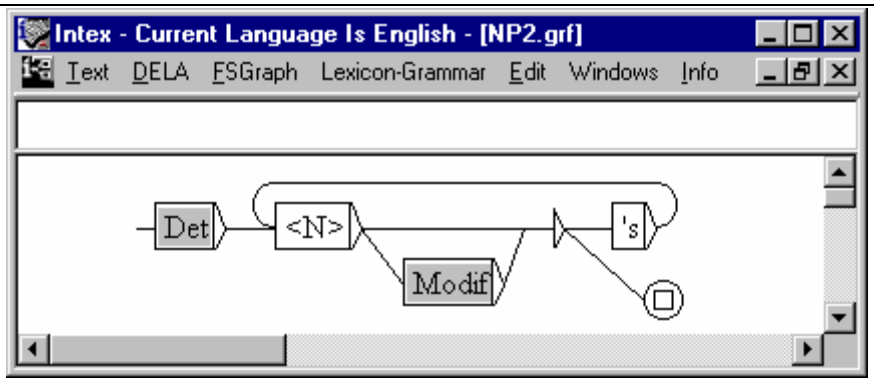


Figure 49. Equivalent grammar, non head recurring

Grammars with tail recursion

Consider the following grammar:

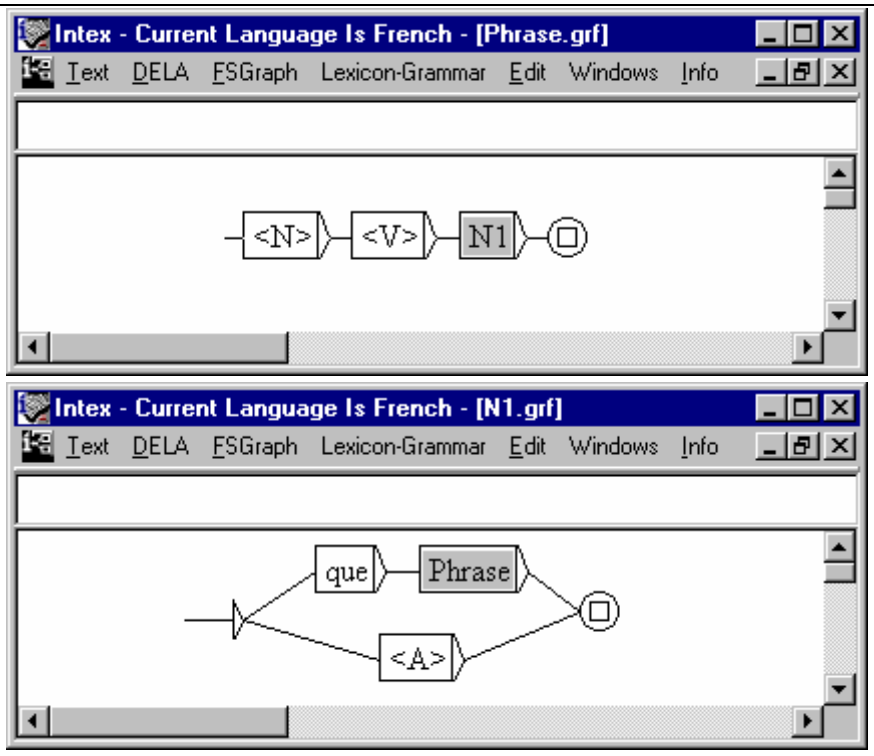


Figure 50. A Grammar with Tail Recursion

This grammar is indirectly tail recursive, because the graph Phrase contains a reference to graph N1, which itself contains a reference to the graph Phrase. This grammar recognizes sequences like the following:

Jean pense que Marie veut que Paul sache que Lisa dit que Luc est idiot

There as well, the tail recursion can automatically be removed from the grammars. For example, the grammar Phrase is equivalent to the Phrase2, which is a finite state grammar:

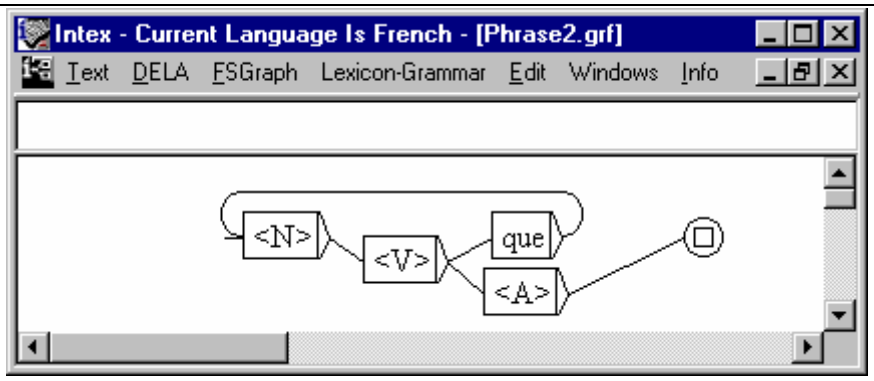
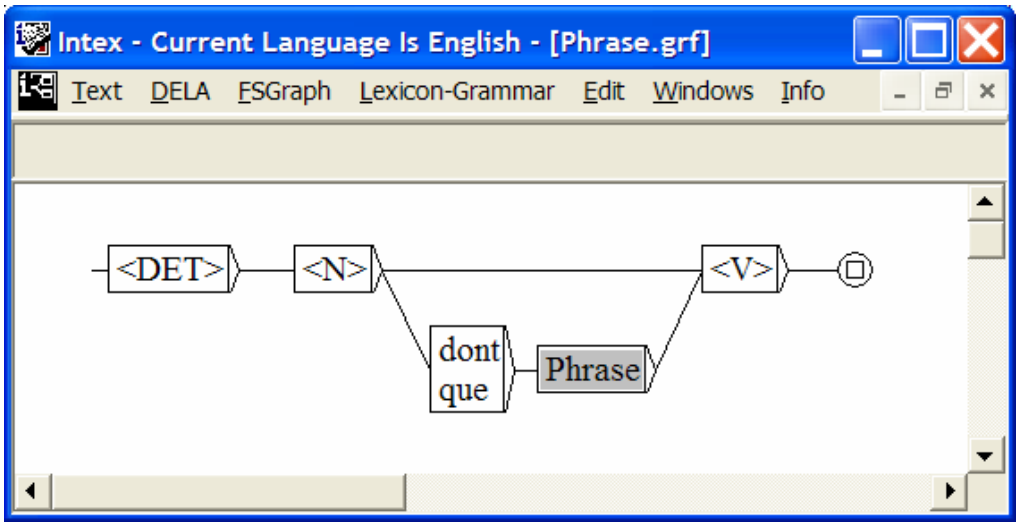


Figure 51. An equivalent finite state grammar

Grammars with middle recursion

Contrary to the preceding forms of recursion, grammars with a more general recursiveness (neither head nor tail), as in the following grammar, are strictly CF (an equivalent non-recursive grammar cannot be constructed for these forms):



This grammar takes into account relative prepositions, which may occur to the right of the subject, as for example:

Le chat (que Paul a acheté) est blanc,
Le chat (que le voisin (dont tu m'as parlé) a acheté) est blanc

This grammar is, however, an insufficient description of the phenomenon that we can observe in French. Truthfully, in the embedded relative clause, there is no complete phrase as the following impossible construction shows:

** Le chat (que Paul a acheté un chien) est blanc*

The clause introduced by *que* must not contain a direct object, while the clause introduced by *dont* must not contain an indirect object introduced by *de*. In more general terms, the graph situated to the right of the relative pronoun should not be Phrase, but rather a graph that represents **incomplete** phrases, the structure of which depends on the relative pronoun in question.

It is interesting to note that natural languages rarely authorize insertions of more than three levels as we can see in the following examples:

Le chat dont tu m'as parlé a faim
? le chat que le voisin dont tu m'as parlé a volé a faim
? le chat que le voisin dont l'enfant ne travaille plus a volé a faim
** le chat que le voisin dont la femme avec qui Luc a travaillé est partie a volé a faim*

It appears then that natural languages are far less recursive than is generally held!

In theory, we cannot construct a finite transducer equivalent to this grammar. If you apply the command Graph > Compile FST, INTEX will compile an **approximation** of the grammar, limiting the number of possible insertions arbitrarily at 10 levels.

If we construct the transducer "**S.fst**" from the preceding graph S (**FSGraph > Compile FST**), the sequences of less than 10 "a" are correctly verified, but sequences of more than 10 "a", equivalent or not, will all be refused. Similarly, INTEX can compile the transducer "**Phrase.fst**" corresponding to the preceding graph, with the restriction that only phrases of less than 10 relative clauses will be correctly treated (which is a reasonable limit!).

Thanks to this limitation, INTEX can treat the graphs containing recursion while staying in the confines of finite state automation and transducers.

The project to construct widely applicable descriptions of natural languages within the boundaries of finite state machines is therefore not unrealistic.

8.2. Enhanced Transducers

Enhanced Transducers are finite transducers that use internal variables to identify and place parts of recognized sequences (affixes). This function is not unlike a similar

function within programs like UNIX type SED. Note that the tokenizer and the morphological analyzer within INTEX are based on the use of enhanced transducers or transducers with internal variables (cf. Chapter 12).

Below we'll give examples of syntactic applications. For example, consider the following graph:

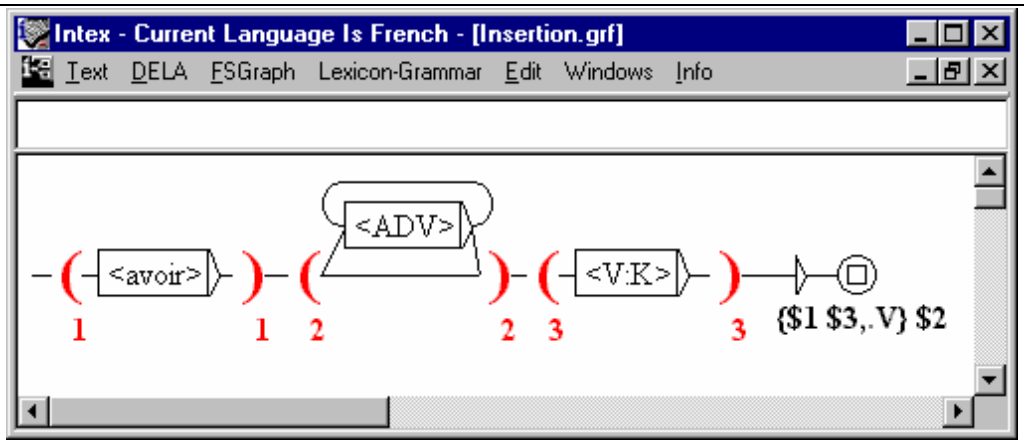


Figure 52. Displacement of an adverbial insertion

To enter the parentheses around the affixes to be held in memory, use the two tags `$(` and `$)`. INTEX will automatically renumber the variables and will alert you of any cases of incoherency.

If we apply the graph to the following text:

```
Luc n'a pas souvent mangé ce plat
```

The sequence "`a pas souvent mangé`" is identified and the variables $1, $2 and $3 are associated to the respective values:

```
$1 = "a" ; $2 = "pas souvent" ; $3 = "mangé"
```

If we apply this transducer in REPLCE mode, the text becomes:

```
Luc n'{a mangé,.V} pas souvent ce plat
```

Here is another example of a transducer:

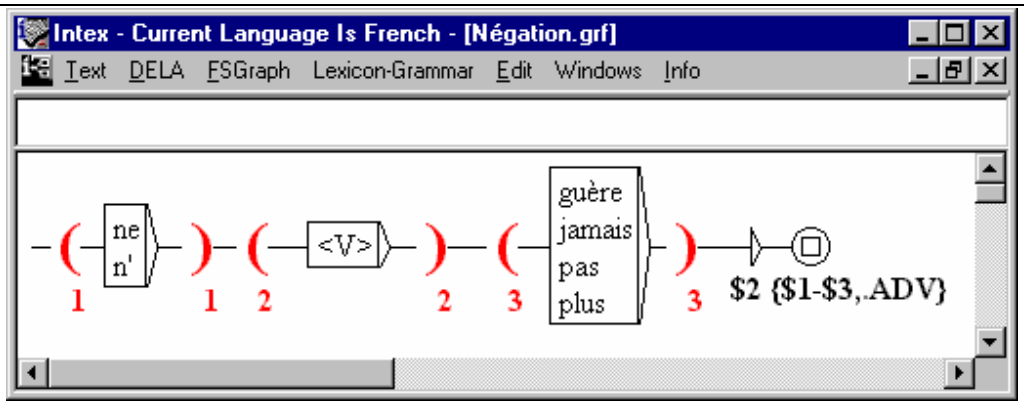


Figure 53. Negation

If we apply this second transducer to the preceding result we will get the final result:

```
Luc {a mangé,.V} {n'-pas,.ADV} souvent ce plat
```

The use of variables in transducers allows us to perform:

-- erasures elements: for example, the substitution: **$1 $2 $3 / $1 $3** allows us to erase the sequence stored in memory with the tag **$2** in the context of **$1** (to the left of) and **$3** (to the right of) the text;

-- insertions: for example, the substitution: **$1 $2 / $1 blabla $2** allows us to insert the text **blabla** between the sequences **$1** and **$2** ;

-- duplications: for example, the substitution : **$1 $2 $3 $4/ $1 $2 $3 $2 $4** allows us to copy the sequence **$2** to two different locations in the context **$1** (to the left of), **$3** (in between) and **$4** (to the right of) the text;

-- permutations: for example, the substitution : **$1 $2 $3 $4 $5 / $1 $4 $3 $2 $5** allows us to change the respective positions of **$2** and **$4** in the context defined by the sequences **$1** (to the left), **$3** (in the middle) and **$5** (to the right).

The use of such transducers in sequence gives INTEX the power of a Turing machine.

8.3. Recursive Transition Networks (RTNs), and Enhanced RTNs

INTEX can also apply **RTN**s (Recursive Transition Networks), which are transducers, which may contain references to embedded graphs (RTNs are to CF grammars, what transducers are to automation).

Finally, these RTNs may themselves contain variables, which would then make them **Enhanced RTNs**.

IV. Text processing

Up until now, we've been working with the file "`Portrait of a Lady.snt`". This file is in INTEX format, which means that it contains linguistic information: the length of the phrases has been limited, the system dictionaries have already been applied to the text, and a series of local grammars have already been applied to eliminate ambiguity.

We will now see how to work with raw text files, or files that come directly from word-processing programs or pulled from the INTERNET. In Chapter 8, we will describe the required file format to prepare a file for the INTEX programs. In chapter 9, we will see how to use transducers to put the file in a linguistic structured format: we will first look at the grammar involved in recognizing phrases then the graphs used to standardize the spelling within the texts.

Chapter 9. TEXT FORMAT

Before attempting linguistic analysis, we must "prepare" the text for analysis, which amounts to formatting it according to INTEX standards. This preparation, however, is based on certain basic hypotheses concerning the format of the text in question.

9.1. Required format for text files

INTEX allows us to work with any language for which the alphabet can be coded on 8 bits (less than 256 characters). The alphabets, the dictionaries and the grammars of INTEX are coded in "Windows ANSI" format, or ASCII extended to 8 bits.

NextStep/OpenStep

If you possess a text coded in NextStep/OpenStep format, you can use the program "`next2iso.exe`" (available in the file "`Intex\App`") to convert the text to Windows ANSI format:

```
c:> next2iso TexteNextStep TexteAnsi
```

Word Processing

Word processors allow anyone to build files in Windows ANSI format from texts stored in memory. For example, in Microsoft Word, click on "**File > Save as**"; under the file name you will see the file format which, by default, will be " **Word**

Document" (with the file extension "`.doc`"); you can replace this format by selecting the format "**Text only**" (extension "`.txt`").

Internet browsers

If you pull a text from a web page, you must also save the page as a "**text only**" document. Most browsers allow this type of operation. For example, in Netscape, click on "**File > Save as"** ; beneath the file name appears the default format ("**HTML**") ; choose "**Text only**".

Publishing

If you are unable to save the text from a given application (word processor, web browser, electronic book, etc.), a simple "**cut and paste**" from your application to a text editor such as Notepad will allow you to convert the file to Windows ANSI format.

The Windows ANSI code allows representation of 256 characters; certain of these characters are control characters (codes 0 to 31); others are special characters for INTEX (for example spaces and digits). Practically speaking, INTEX can process languages that have up to 255 characters (counting capital letters, diacritical marks and punctuation).

Note: **Letters** are the characters contained in the alphabet of a language. The alphabet itself is a simple file stored in the file-folder of the language. All characters that are neither letters nor digits are called **separators**.

9.2. Problems with alphabets

Languages with a large alphabet

Currently, INTEX is unable to process alphabets that contain more than 255 letters and separators. A future version of INTEX (NooJ) will deal with any text encoding, (including UNICODE), which will resolve all problems of representation.

Note however that there are two possible short-term solutions:

- Often there exists transcriptions of these languages (more or less phonetic) which use the Latin alphabet, perhaps somewhat extended; INTEX is able to process transcribed texts if the alphabet used for the transcription contains less than 255 characters;

- Languages with long alphabets often use another alphabet, used to enter texts using keyboards of reasonable size. We are then able to compose each character from the “rich alphabet” using several characters from this alphabet. For example, Korean characters are generally composed from two consonant symbols and on vowel symbol. We can therefore contemplate working with the small alphabet (thirty or so letters, instead of several hundred letters from the rich alphabet); INTEX can process texts with this representation. Note that this transcription can be automatically done both ways (from a raw text to INTEX; from INTEX to resulting texts).

Languages without spaces

Certain languages do not have separators between their words; for example Thai or Chinese. A text therefore looks like an uninterrupted series of letters, at the best of times, delimited with periods (to indicate the end of sentences).

At present, INTEX cannot process series of more than 80 letters without spaces. In order to analyze texts written in languages without spaces, the more simple solution is to automatically add a space after each letter. After that, all INTEX programs will work correctly but certain conventions must be understood:

-- Each phrase will then be viewed by INTEX as a sequence of simple forms, each of which will have a length of 1;

-- the program used to recognize compound words would therefore accomplish the break-up of text into words: to a certain degree, all words of more than one letter in the language are compound words for INTEX.

ASCII 7 bit Code

ASCII 7 bit code (also called ASCII) only represents 128 characters: it does not contain letters with accents.

Word Processors that use ASCII code (for example Latex or HTML) use sequences of characters to represent accented letters; as an example, the letter "é" is represented by the sequence “`e\'`” or “`{e\acute}`”.

INTEX cannot process alphabets in which letters are coded on more than one character. If you want to parse such texts, you’ll first have to convert them to the default Windows ANSI format, by replacing each multi-character sequence by the corresponding accented character. A command such as the “`sed`” will do:

```
c:> sed -e 's/{e\acute}/é/g' TexteAscii TexteAnsi
```

Alternatively, the same could be achieved by performing a series of manual substitutions or by using a macro within the word processor.

9.3. Lines and paragraphs

Paragraph changes or line changes.

We will now see that the automatic recognition of sentences or phrases in a text is partially founded on the notion of the paragraph. The ASCII codes "NEW LNE", (also called "Linefeed") and "CAR RET", noted "\n\r" in C language and "^p" within Microsoft Word, are considered paragraph change markers by the grammar rule responsible for the delimitation of the text into sentences.

This convention is used by numerous word processors, since the line change generally comes into play automatically. (this varies depending on the size of the margins and the font in use.).

It is possible, that within the text you pulled (for example by using the Cut & Paste function within an Internet Browser or an E-Mail program), that these characters will correspond to a line change command as opposed to a paragraph change. In order to define the sentences / phrases in such a text-file, there are two possibilities. Either...:

- you can modify the text file, replacing two line changes by a paragraph change, and a line change by a single space; for example, in Microsoft Word, this operation can be done in three steps: (1) Replace "^p^p" with "XWXW", (2) Replace "^p" with " " (a space), and finally (3) "XWXW" by "^p".

- or you can mark the paragraph changes in the text, for example by inserting, at the start of each new paragraph, a special character such as "@", then modify the graph used by INTEX to recognize the sentences of a text, so as to ensure that it takes into account the new paragraph change marker ("@"), as opposed to the line change.

Note that there exist numerous application for which text need not be defined into sentences: for example, in automatic documentation, we may prefer to analyze legal texts article by article; in literary analysis, we'd want to analyze poems rhyme by rhyme, etc.

Marked Paragraph changes

In certain word processors, the codes "`NEW LNE`" or "`NEW LNE, CAR RET`" are considered as spaces, without any significance with respect to formatting; the paragraph change is therefore marked with a special code, such as "`\par`". To place such codes into a format acceptable to INTEX, we need only to replace the "`NEW LNE`" or "`NEW LNE, CAR RET`" codes by a space, and the sequence of codes "`\par`" by the "`NEW LNE`" code.

9.4. Meta-characters

Forbidden Characters

When INTEX attempts to perform linguistic analysis on the texts in question, it will insert linguistic information in the form of codes written between "accolades" ("{ }"). For example, the code "`{S}`" represents the definition of a sentence, and the code "`{la,le.DET:fs}`" represents the feminine, singular determiner *la*.

Since the two characters "`{`" and "`}`" have a special significance for INTEX INTEX, we recommend changing these characters, in the texts to be processed, with other characters (for example, "`[`" and "`]`").

The character "#" has an internal significance for INTEX; we recommend replacing it with another character as well, such as "£".

The character corresponding to the code 0 (noted as "\0" in C) has particular significance for all programs written in C language (among others, INTEX). We recommend replacing it with another character.

The codes which correspond to the ASCII values between 1 and 31 are generally used as control codes to represent instructions to the printer (for example, change of page), calls for notes, markers for indexed words, etc. The grammars that are internal to INTEX do not take these codes into account; for example, the grammar rule that recognizes sentences does not take into account potential calls for notes or page changes.

We therefore recommend either eliminating these control codes or modifying the INTEX grammars or dictionaries that you use.

Special Characters

The ten special characters used in the rational expressions:

```
< > { } + " \ * ( )
```

and the nine special characters used in the graphs:

```
< > { } + " \ / :
```

can appear in texts; remember however that in order to search for the sequences that contain them, you must “protect” them in one of two ways. :

-- each of these characters must be preceded by the special character "\" ; for example, to look for the sign "+" in a text, you would enter the following rational expression:

```
\+
```

-- all of these characters, with the exception of quotation marks, may be placed in between quotation marks; for example:

```
"+"
```

Note that the quotation marks can be used to perform an exact search; for example, the following expression recognizes only the form *table*, but not the forms *Table*, *TABLE* or *TaBle* :

```
"table"
```

Caution: the quotation mark cannot be placed between quotation marks.

In the graph editor, the character “:” acts as a special character only when it is placed at the beginning of a term, in the tag or label of a node. In rational expressions as in graphs, the character “>” is considered a special character only when it is preceded by the meta-character “<”.

Particular coding

Certain applications use more or less “exotic” codes to represent text; this coding can have repercussions within the standard processing of INTEX:

- If a text was written entirely in lower case letters, the default grammars of recognition for sentences and proper nouns within INTEX will not work with correct results; the user will have to modify them or create their own grammars;

- If a text contains inserted information, for example in the form of XML markers, consultation of INTEX dictionaries will be disturbed: on one hand, these markers risk being treated as unknown words; on the other, the presence of one of these markers within compound words or frozen expressions will hinder their recognition.

Unless you perfectly master the problems that can arise from the application of dictionaries and grammars to texts of this type, we recommend bringing these texts back to a more standard form of coding.

9.5. Verifying the format of a text

After having obtained a text in Windows ANSI format, you must first verify that the file does not contain any forbidden characters (ASCII codes from 0 to 31, the characters "#", "{" and "}"), you must also verify whether or not the alphabet of the text corresponds exactly with that of the active language.

To do this, bring up the text file: Choose the menu "**Text > Open…**", then replace the default file type by "**INTEX Delimited Text**" (at the bottom of the window) by the file type "**Windows ANSI Text**" **(A)**.

File types are represented by an extension: "`.txt`" for Windows ANSI, "`.snt`" for files formatted for INTEX. The file "Portrait of a Lady.txt" will appear **(B)**. Select, then open it. (C).

Caution: do not confuse the file extension "`.txt`" (file in Windows ANSI format) with the file extension "`.snt`" (file in INTEX format) on one hand, nor with the folders "`_txt`" and "`_snt`" on the other (the folders where information about each file is stored).

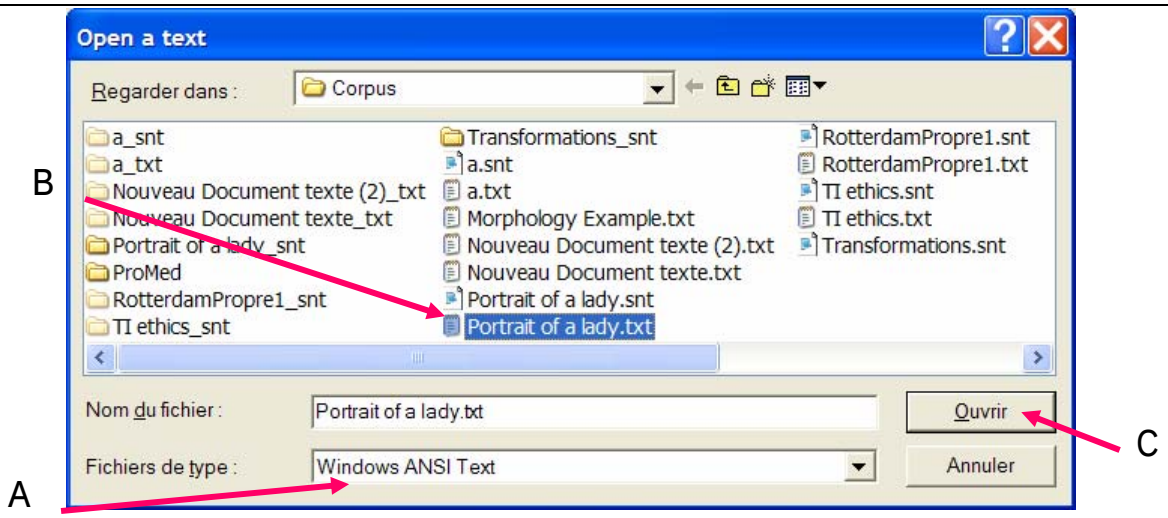


Figure 54. Opening a text file in Windows ANSI format

It is possible that the file extension not be visible. In this case, click on the **Tools** menu in the **My Computer** window, choose **Options** then on the **View** tab. Scroll down until you can see whether or not the check box corresponding to "**Hide extensions for known file types "** is checked.

INTEX will then prompt you to standardize the text ("*INTEX likes pre-processed texts. Proceed?*") ; answer "**No**" for the moment; the window "**Attention / Caution**" will explain the system limitation if you do not standardize the text; Click "**OK**" ; INTEX will then index the text.

After a moment or two the "**Tokens list**" window will appear; this window is the result of the indexation of the text and lists for you the 100 most frequently occurring lexemes (*tokens*) in the text.

Click on the button "**Show Chars**". This will give you the list of all characters present in the text, divided into three classes: *delimiters, digits and letters*. Each character is presented with its Windows ANSI code in parentheses, then with its frequency of occurrence.

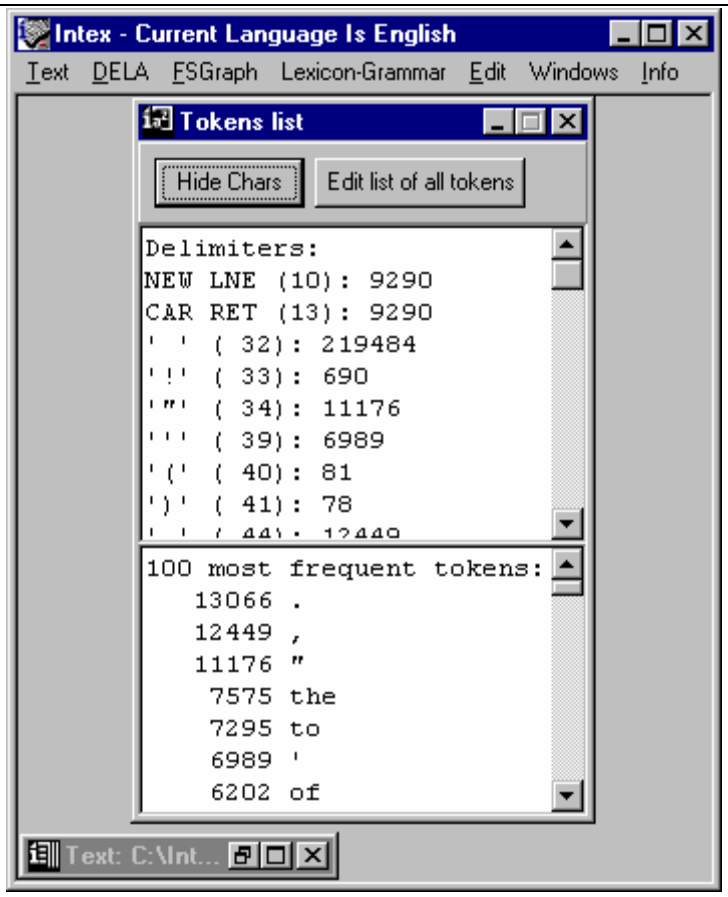


Figure 55. Result of the indexation of a text

- You must verify that the forbidden characters "#", "{", "}" and that of the code 0 are not present in the text;
- If certain control characters (for which the ASCII value is between 0 and 31) are present, you will either have to remove them from the file or ensure that you understand their role (for example, to mark indivisible spaces or page changes), and that you write your INTEX queries and grammars accordingly;
- If the character "CAR RET" is present, ensure that its frequency is identical to that of the character "NEW LNE". In Windows, there are two ways to indicate or represent a paragraph change: either by the character "NEW LNE", or by the sequence "NEW LNE, CAR RET"; all other cases reflect a use that deviates from standard Windows ANSI code;
- All the letters or the language are in fact classed as such.

The last point is important: for example, you will see that numerous French texts contain accented upper case letters or ties / ligatures that are not described in the default French alphabet provided in INTEX. If you encounter such characters, you can either:

- replace these characters by their transcriptions which correspond to the default alphabet (for example, you'd replace "æ" by "ae", "Ê" by "E", etc.) ;

- or modify the alphabet (the **Alphabet** file is stored in the folder of the active language, in your personal INTEX folder) and add these characters.

Chapter 10. PRE-PROCESSING A TEXT

In the preceding chapter, we discussed the format required by INTEX in order to process the text-file at the character level. We will now talk about the format required for INTEX to be able to process the text-file at the word and sentence level.

The process of changing a text-file in Windows ANSI format, into a text-file in INTEX format is called Pre-processing. It consists of a linguistic pre-analysis, the goal of which is to divide the full text into more manageable textual units (generally, sentences), and to standardize the spelling of certain forms that are not addressed in the lexical module (e.g. "aujourd" in French, or "doesn" in English). This is accomplished in three steps:

- segmentation of the text into sentences will be accomplished by the application of a transducer in "MERGE" mode; this transducer will insert the defining marker for sentences "{S}";
- processing words or non-ambiguous compound expressions consists of tagging these forms, in such a way as to eliminate non-autonomous constituents of the text; this is done by a simple consultation of the dictionary of non-ambiguous forms;
- the standardization of certain contracted or elided forms will be done by the application of a transducer in "REPLACE" mode.

We will now work on a Windows ANSI text; you can take one of your own texts (if you've ensured that its format is correct), or use the Windows ANSI version of "La femme de trente ans".

Use the "**Text > Open**…" command. At the bottom of the dialogue box **(A)**, you can choose the file type: choose "Windows ANSI" instead of the default "**INTEX Delimited Texts**", and then choose the file "`La femme de trente ans.txt`" (exactly as in the preceding chapter, when you wanted to verify the file format).

INTEX then asks if you'd like to pre-process the text ("*INTEX likes pre-processed texts. Proceed?*"); this time, answer "**Yes**" ; a new dialogue box will open:

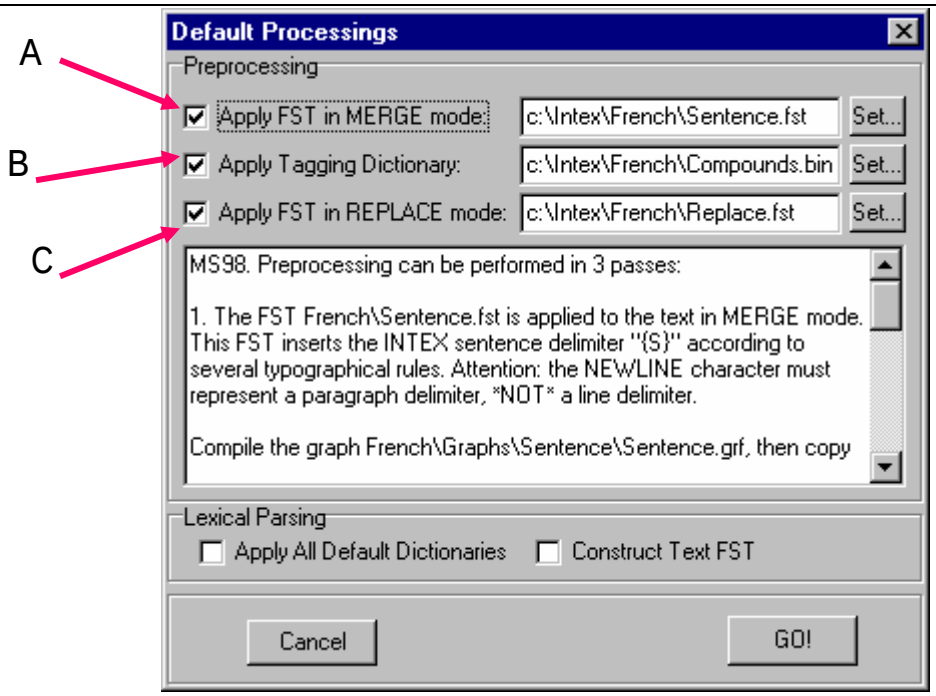


Figure 56. Preprocessing a text

Preprocessing the text consists of three steps:

- INTEX applies a transducer in "MERGE" mode. **(A)**; then
- INTEX applies a dictionary of non-ambiguous compound words **(B)**;
- Finally, INTEX applies a transducer in "REPLACE" mode **(C)**.

Ensure that the three check boxes are checked "☑" in the "**Preprocessing**" section, as well as the two check boxes in the "**Lexical Parsing**" section. Click "**GO!**". INTEX will perform the three steps of pre-processing. After a moment, the text is pre-processed and the corresponding INTEX file is created: "**la femme de trente ans.snt**".

To show the tags or the linguistic information, click on "**Display tags**" located at the top left side of the text window. We'll now describe the three steps in more detail.

10.1. Segmentation of the text

The transducer used to segment the text into sentences is the file "`Sentence.fst`" found in the folder of the active language. This transducer is compiled from the graph:

`Graphs\Preprocessing\Sentence\Sentence.grf`

You can see and edit this graph (more precisely, the version of this graph stored in your personal folder!).

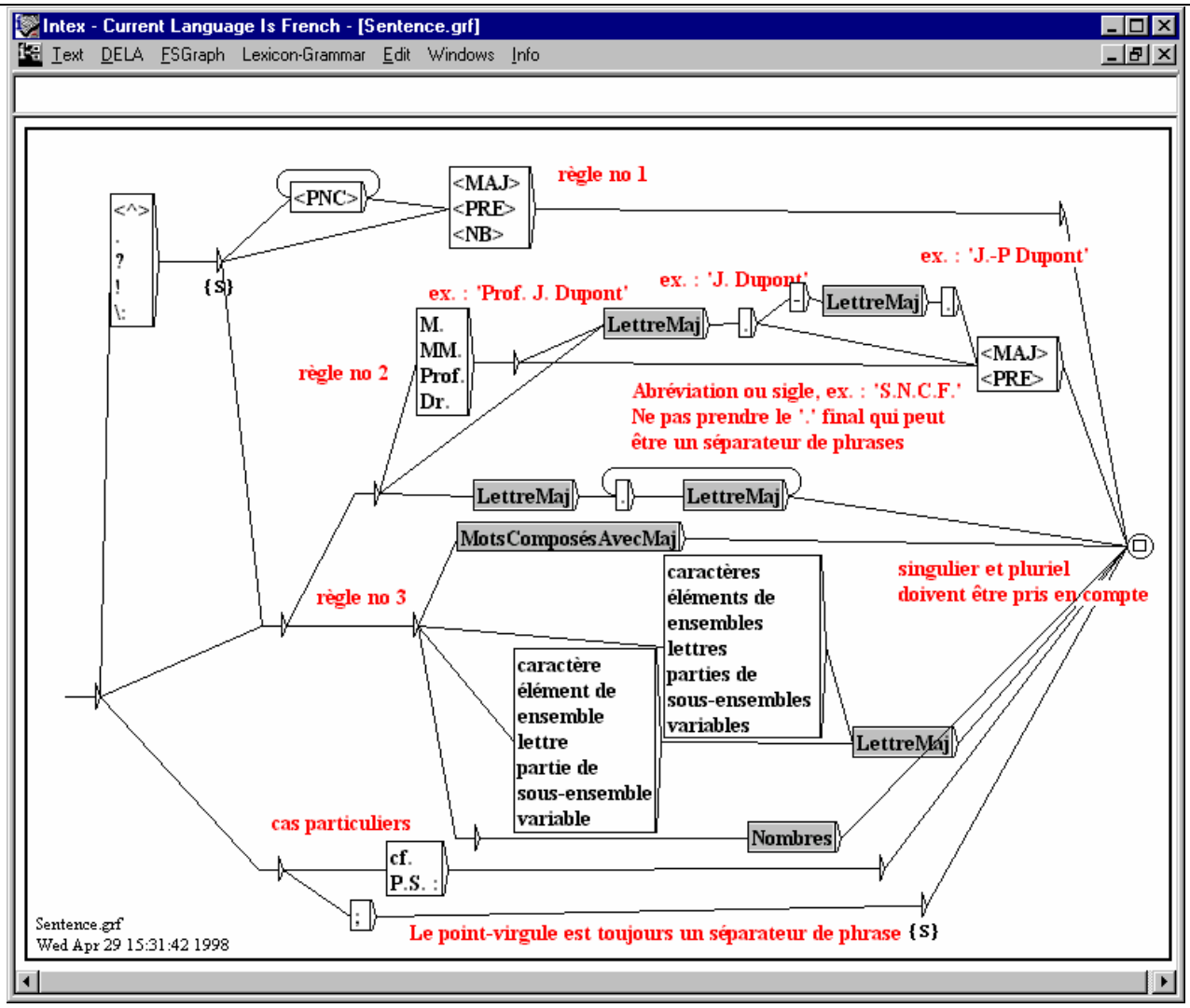


Figure 57. The Sentence *graph*

-- Rule 1 (at the top of the graph): if a beginning of paragraph is encountered (`<^>`), a period, a question mark, an exclamation mark, or a character "`:`", potentially followed by any number of separators (`<PNC>`), followed by a capitalized word or a number, the tag `{S}` is inserted after the period. For example, the text:

```
Comment allez-vous ? "Bien, merci". 5 sur 5 !
```

becomes:

```
{S}Comment allez-vous ?{S} "Bien, merci".{S} 5 sur 5 !
```

-- Rule 2: if we encounter "`M.`", "`MM.`", "`Prof.`" or "`Dr.`" followed by a capitalized word (for example: "*M. Dupont"*), INTEX will not insert an "{S}" tag. The embedded graph LettreMaj identifies the 26 words that begin with a capital letter. For example, the following text will remain unchanged:

```
M. Dupont, C. Duras et le Prof. M. Javert
```

Acronyms like: "**S.N.C.F.**", are not separated, but nothing hinders the final period from also indicating an end of sentence. Note then, that rule 2 acts, to a certain extent, as an exception to rule 1.

-- Rule 3: We've seen in the preceding case, that we cannot insert a sentence separator between "`C.`" and "`Duras`". But in certain particular scenarios, for example:

```
Luc utilise la vitamine C. Duras préfère le pain.
```

a sentence separator must be inserted. The sequence "`C. Durand`" should not be considered an expression: it must therefore be, that the sequence "`vitamine C`" be recognized first. Compound words, of which the last constituent is a capital letter are described in the embedded graph MotsComposésAvecMaj. We can therefore say that rule 3 acts as an exception to rule 2: these compound words are not separated (the transducer will not insert the `{S} tag`), but after having identified one of these words, INTEX continues to process the text, which means that the sequence "`. Duras`" is identified thanks to rule 1. The preceding text will then become:

```
Luc utilise la vitamine C.{S} Duras préfère le pain.
```

We've added to the description, certain semi-frozen expressions from mathematical language.

-- Some particular cases have also been taken into account: numbers (which contain a period), the period-comma (separating two sentences, the second of which may not necessarily begin with a capital letter), and some abbreviations.

This graph provides satisfactory results when it is applied to journalistic texts or novels, if the text-file is correctly formatted (as it relates, in particular, to the question of paragraph change representation). More than 99% of sentences will be correctly separated. One error that is regularly seen by this graph occurs when certain names are abbreviated "à l'anglaise" (with 2 given names abbreviated as opposed to a single compound name), as in:

```
Le réalisateur C. B. De Mille a fait de grands films
```

The **Sentence** graph produces the following erroneous result:

```
Le réalisateur C. B.{S} De Mille a fait de grands films
```

because "C. B." would be treated as an acronym (such as "S.N.C.F."). Note that in the following case (more common in French), the graph produces a correct result:

```
Ils écoutent la C. B.{S} Je préfère le téléphone portable
```

The solution would be to describe abbreviated (or unabbreviated) names of Anglo-Saxon personalities in a local grammar.

There exist several applications wherein we prefer not to process texts sentence by sentence. For example, in the study of poetry, the textual unit would be the verse or the stanza; for information searches in legal texts, the textual unit would be the article; for dictionary analyses, it would be the definition of a term, etc. In all of these cases, do not apply the transducer "**Sentence.fst**" suggested by INTEX; rather, write your own grammar, to insert the textual unit separator there where it is most useful.

10.2. Tagging non-ambiguous compound words

The second stage of pre-processing consists of identifying and tagging the non-ambiguous compound words that contain non-autonomous constituents. For example, consider the following words:

```
aujourd’hui, parce que, clopin-clopant
```

From a formal point of view, these three sequences each contain a separator: the apostrophe is not a letter, it is therefore a separator; the space and the hyphen are also separators. When it comes to indexation of the text, INTEX does not differentiate between these three sequences and the following three similar sequences:

```
l’arbre, la maison, dit-il
```

In order to avoid allowing the system to process the following non-autonomous forms as lexemes (these forms are non-autonomous because in no context will they ever occur simply as…):

```
aujourd, parce, clopin, clopant
```

we can **tag** the compound words *aujourd'hui*, *parce que* and *clopin-clopant*, in other words, we replace these compound words in the text with the corresponding lexical entries:

```
{aujourd'hui,.ADV} {parce que,.CONJS}
{clopin-clopant,.ADV}
```

From there, the non-autonomous forms like "aujourd" have literally disappeared from the text; the lexeme, i.e. the minimal unit of processing hence becomes the entire compound word.

In order to verify that this disappearance has occurred, look for the form "`aujourd`" in the text "**la femme de trente ans**": you will not find a single occurrence, on the other hand, if you look for the tag `{aujourd'hui}`, you will find it.

Tagging compound words with non-autonomous constituents presents two important advantages:

- on one hand, we make the non-autonomous forms disappear from the text; we can then benefit from the analysis of a certain part of the text (since we've replaced the compound words by their corresponding lexical entries);
- on the other, since the non-autonomous forms have disappeared from the text, there is no need to look for them in the dictionary of simple words; it is therefore un-necessary to code the non-autonomous words in the dictionaries of simple words (the symbol of category "**X**" disappears from the DELAF).

For French, the dictionary "**Compounds.bin**" (selected by default, stored in the folder of the active language) contains approximately 1 000 compound words of this sort. Naturally, you are able to construct and apply you own dictionary of non-ambiguous compound words, in the DELAF "`.dic`" format, or in the compressed form "`.bin`", for example, to tag the technical terms or different references to products (in the area of automatic documentation).

There probably exist text analysis applications for which one would not want to eliminate the non-autonomous words from the text, nor process the non-ambiguous compound words as minimal units. For example, if you want to study certain plays on words or study the distribution of syllables in poetry, etc. In these cases, do no select the second check box.

10.3. Rewriting of "deviant" forms

The third phase consists of identifying and rewriting certain non-ambiguous, *deviant* sequences, as for example, in French:

au ("à le"), auxquelles ("à lesquelles"), n’ ("ne"), etc.

or in English:

cannot ("can not"), doesn’t ("does not"), I’ll ("I will"), etc.

These *deviant* sequences are deemed such given that they present a spelling anomaly that could unnecessarily interfere with several analyses:

- from the point of view of a word count, *au* should be considered as two words, without which, the preposition *à* and the determiner *le* risk not being counted, indexed nor even identified within the text;
- from the point of view of syntactic analysis, *au* will find itself somewhere in between a noun phrase (which begins with a determiner), and a prepositional phrase. Rather than weigh down the grammars by forcing them to take into account certain prepositional phrases wherein the preposition and the determiners are fused together, it is far more simple to separate *au* into its constituent "`{à,.PREP} {le,.DET:ms}`".

The third phase consists, therefore, of applying a transducer (by default, "**Norm.fst**" stored in the folder of the active language) in "REPLACE" mode in order to rewrite these sequences of *deviant* forms into more standardized forms.

The difference between the second and third phases of the pre-processing is that, during the second phase, we could replace a sequence of several simple forms with lexical entries whereas, in the third phase, we can replace a sequence of one or more simple forms with a sequence of one of more simple forms, or lexical entries. The third phase is therefore more "powerful" than the second (but it is also requires more time to complete).

Caution: We can only carry out this type of substitution for non-ambiguous sequences.
For example, the simple form *des* can, in certain cases be the contraction of the preposition *de* and the determiner *les*, as in:

```
Luc rêve des vacances
```

This being said, we cannot replace every occurrence of "`des`" in the text, by the sequence "`{de,.PREP} {les,le.DET:mp:fp}`" since, in the majority of

cases, *des* is not a contraction! The graph Norm.grf (stored in the folder "`Graphs\Norm`") calls the two graphs **Elisions** and **Contractions**:

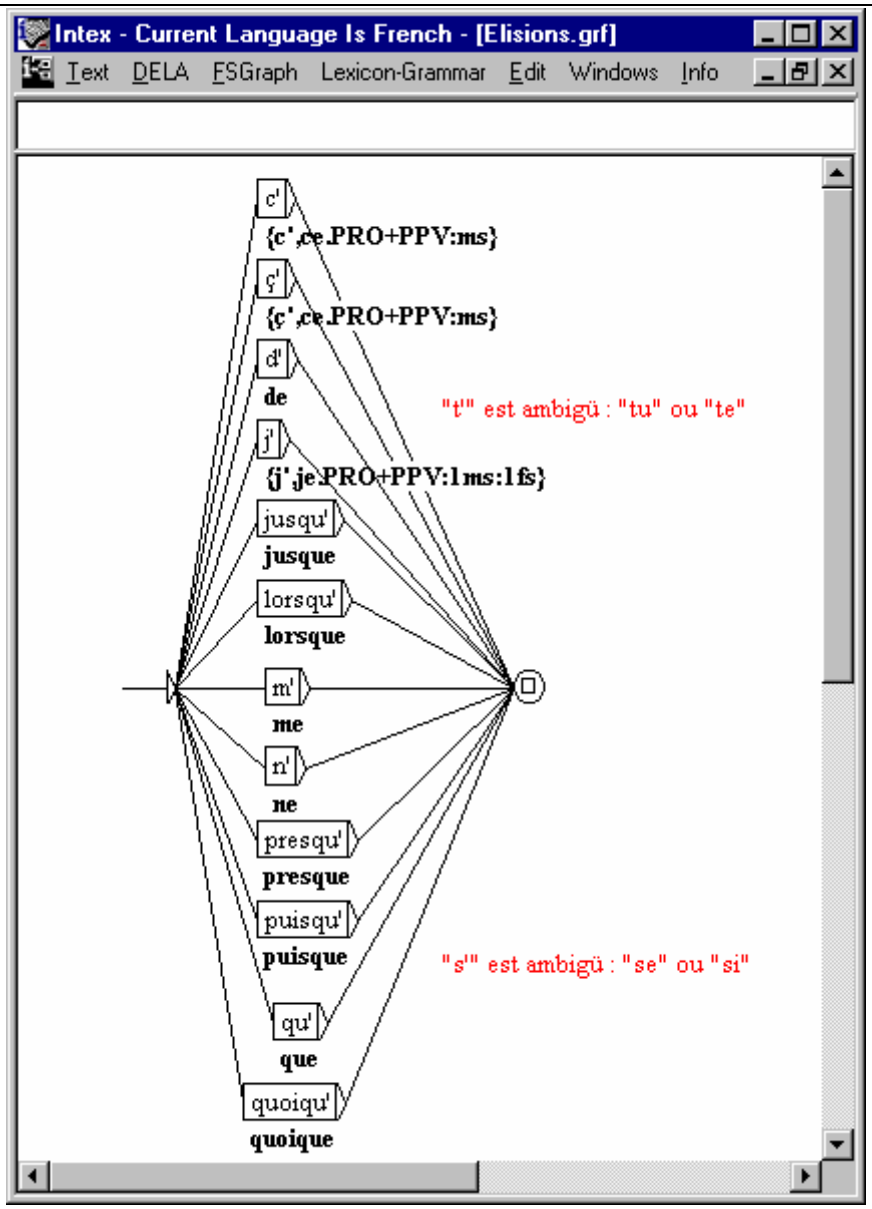


Figure 58. Resolving certain cases of elision

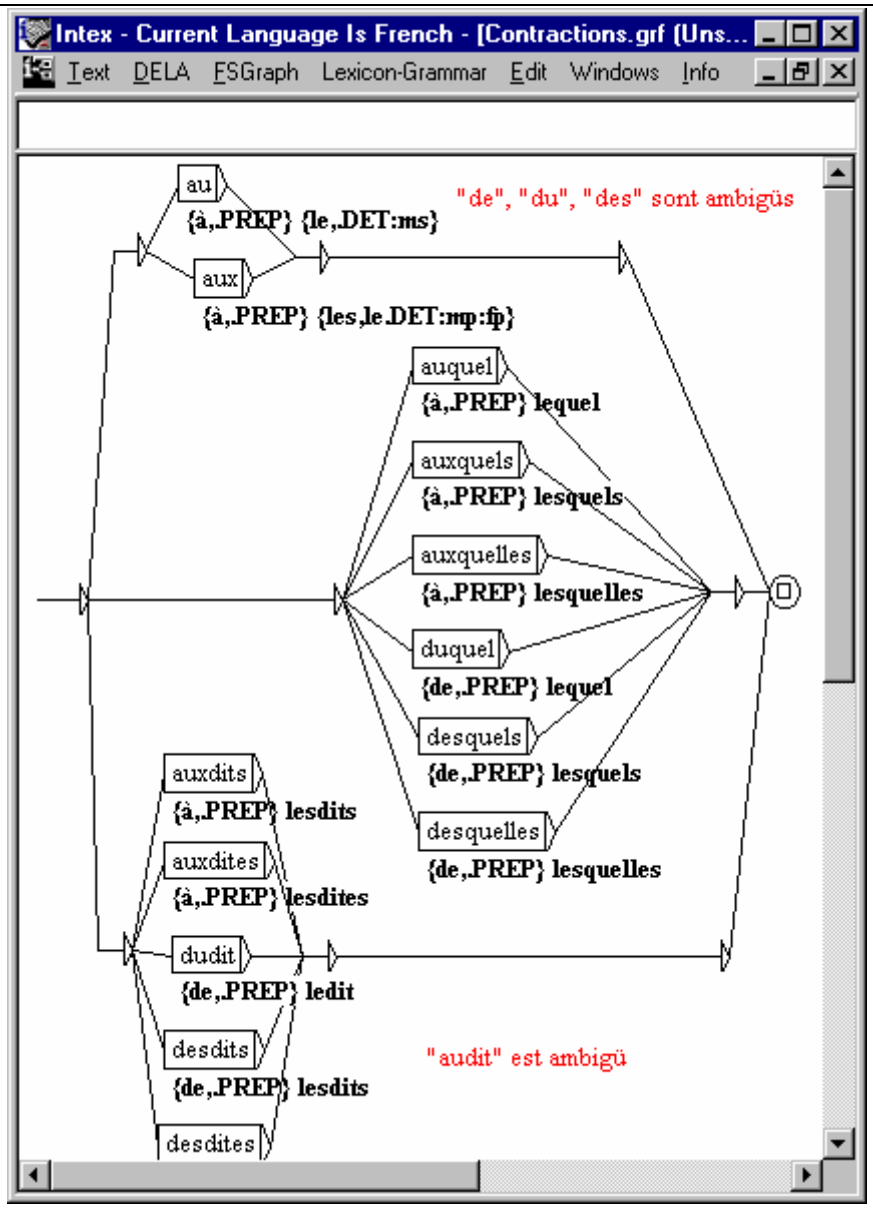


Figure 59. Resolving certain cases of contraction

We must remember that only non-ambiguous elisions and contraction are treated here; for example, *du* can be the contraction of the preposition *de* and the determiner *le*, but also the partitive determiner. Introducing potential errors into the text is out of the question: ambiguous elisions and contractions are therefore left as-is. We will see later how to represent ambiguous elisions and contractions in the graph of the text, and later how to resolve this ambiguity.

For specific applications, it may be interesting to modify or add other substitutions to the pre-processing; for example, in automatic documentation, we may wish to standardize the spelling of names referring to products; for INTERNET applications, we may wish to replace links or addresses by typed markers, etc. In such cases, do not hesitate to modify the **Replace.grf** graph which is stored in the folder:

`Intex\French\Graphs\Preprocessing\Replace`

Then, compile your new graph (**FSGraph > Tools > Compile & Determinize FST**), and copy the resulting **Replace.fst** into the folder of the active language.

Inversely, there exist applications for which we do not want to standardize contractions or elisions; for example, if you want to analyze the text for the purpose of vocal synthesis! In these cases, don’t perform the third stage of pre-processing.

V. Lexical analysis

Working from the alphabet of the active language, INTEX is able to process **simple forms**. Now, we will go further: we want to be able to process any type of **linguistic units**. We see in chapter 11 that INTEX can identify atomic linguistic units in a text by consulting the dictionaries, or by recognizing lexical graphs. Chapter 12 describes the tokenization and morphological module of INTEX.

Chapter 11. LEXICAL RESOURCES

11.1. Lexical units

Lexical units are the linguistic units that make up the atoms of the sentence, or the non-analyzable units of the language. Intuitively, they are words, the meaning or function of which cannot be calculated: One must learn them to be able to use them. From a linguistic point of view, it is invaluable to catalogue them and to describe their properties (generally as a lexicon, hence then name *lexical units*).

From a formal point of view (in terms of the definition of simple forms), we can separate the lexical units into four classes:

Simple Words are the atomic linguistic units that are written as simple forms. For example, *pomme*, *table*.
Morphemes are the atomic linguistic units that are sequences of letters, included in the simple forms. For example: *re-*, *-ation*.
Compound Words are the atomic linguistic units that are sequences of letters and separators. For example: *cordon bleu*, *pomme de terre*.
Frozen Expressions are the atomic linguistic units, which are potentially interrupted sequences of letters and separators. For example: *prendre … en compte*, *ne … pas*.

The terminology used here is natural for romance languages; it is, however, less well suited for the Germanic languages, where we are accustomed to using the term compound word to refer to a sequence of *analyzable* letters without a separator (In

INTEX, compound words are sequences of *non-analyzable* sequences of more than one simple form with at least one separator.) and for languages like Chinese where there are no separators in words. What is important to consider, is that no matter what the language, the linguistic units can be grouped into four classes, which correspond to an automatic recognition program within INTEX.

The linguistic unit recognition program of the first type (simple words from the romance languages), will look for each simple form (sequence of letters between two separators) in the dictionary of simple words. The second type of linguistic unit recognition program will see to break the simple forms into smaller units (*morphemes* in the romance languages). The third type will look for each sequence of one or more simple forms in the dictionary of compound words. The fourth type, will apply certain grammars to identify sequences that do not necessarily resemble the simple forms.

Caution: do not confuse the terms simple word and simple form; a simple form is, by definition, any sequence of letters (that do not necessarily correspond to an atomic lexical unit, for example "*ecthz"* or "*redéstructurabilité*"); a simple word is, by definition, an atomic linguistic unit. DELAF type dictionaries are designed to distinguish between simple words and simple forms (in other words, simple words are simple forms catalogued in DELAF type dictionaries).

The **recognition** of an atomic linguistic unit by the lexical module of INTEX does not imply that the linguistic unit truly appears in the text. For example, the fact that we find the sequence of two simple forms, "`cordon bleu`", in the dictionary of compound words does not necessarily mean that the compound word meaning “a good cook” is indeed present in the text. Such is also the case for morphological analysis and the referencing of the simple word dictionaries: the simple form `repasser` would be considered ambiguous (*Luc repasse sa chemise* vs *Luc repasse par là*). The frozen expressions in particular are often ambiguous with respect to a sequence of simple words: for example the sequence of three simple forms *casser sa pipe* will sometimes be interpreted figuratively as "*mourir*" ("*kick the bucket"*), and sometimes literally as ("*break one’s pipe*").

In INTEX, load a text if needed, the call up the lexical module by clicking "**Text > Apply Lexical Resources**". The dialogue box that appears contains three areas: **Simple Words**, **Compound Words**, **and Frozen Expressions**. The "**Simple Words**" zone also contains the necessary tools to perform a morphological analysis of simple forms.

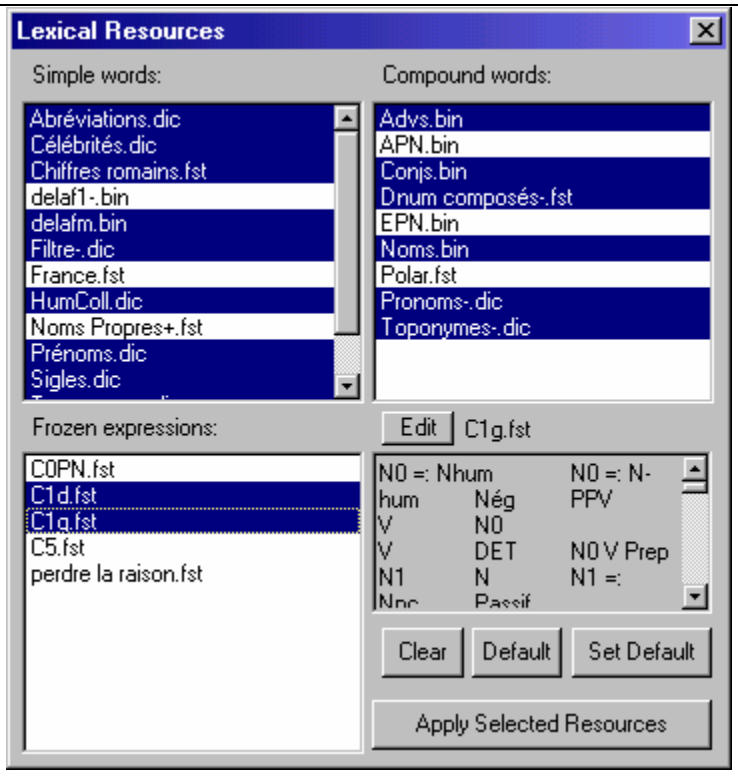


Figure 60. Tools of lexical recognition

The tools used to identify linguistic units (*lexical resources*) are of two types:

- **Electronic dictionaries** are files in DELAF (simple words) format, or DELACF (compound words) format. The dictionaries described in the files with extension "`.dic`" can be edited (you can both see then modify their contents); the dictionaries described in files with the extension "`.bin`" are compacted and their contents cannot be modified.

- **Lexical transducers** ("`.fst`" files) can be obtained either from graphs or from lexicon-grammar tables.

Technically, "`.bin`" type files are in fact lexical transducers; we speak of them as compiled dictionaries because from the point of view of the end user, they always come out of "`.dic`" type dictionaries that were compiled using the **DELA** > **Compress into FST** command.

Dictionaries cannot represent frozen expressions because they are open to insertion. Lexical transducers obtained either from graphs or from lexicon grammar tables, therefore represent them.

11.2. DELAF type electronic dictionaries

In order to process texts written in a given language, INTEX requires a dictionary which houses and describes all of the simple words of that language, with their inflected forms (in French: the conjugated forms of verbs as well as plural and feminine forms of nouns and adjectives).

The main idea behind construction of a DELAF electronic dictionary of French is described by [Courtois, 1990]. On the same principal, several teams have built DELAS-DELAF systems for English, Spanish, Greek, Italian, Portuguese, etc.

We will see in the following section, that DELAF type dictionaries can be automatically built from DELAS type dictionaries that only contain the lemmas of the various forms (in French: the infinitive forms of the verbs and the singular, masculine forms of nouns and adjectives) as well as the flectional codes used by INTEX to automatically build the inventory of corresponding inflected forms.

Generally, the DELAF dictionary of a given language contains all of the inflected forms of the language and associates them with a lemma, a morpho-syntactical code and possible syntactic, semantic and flectional codes. Here, for example, are some entries from the DELAF of French:

```
avions,avion.N+Conc:mp
avions,avoir.V+aux:I1p:S1p
cousins,cousin.N+Anl:mp
cousins,cousin.N+Hum:mp
de,de.PREP
```

The first line represents the fact that the form *avions* is associated with the lemma *avion*; it is a noun (**N**), for which the distributional class is Concrete (**+Conc**); it is in the masculine plural form (**:mp**). The second line represents the fact that the form *avions* is also associated with the lemma *avoir*, which is a verb (**V**), an auxiliary (**+aux**); the form is conjugated in the imperfect tense, with the first person plural (**:I1p**) or in the present subjunctive, also first person plural (**:S1p**). The two following lines represent the two nominal forms *cousins* (animal of human). The last line represents the form *de*, which is identical to its lemma and is a preposition (**PREP**).

When a form can have more than one flectional analysis while being associated with the same lemma, the same category code and the same syntactical and distributional information, we represent this form with only one line (for example, the verbal form *avions* has two flectional analyses); On the other hand, if a form is associated with different lemmas or different morpho-syntactical category codes, or to different syntactical or distributional information, we repeat the form (as is the case of *cousins*).

Let us remind the reader of the codes found in the DELAF dictionary of French:

Code	Meaning	Examples
<A>	*Adjective*	*artistique, bleu ciel*
<ADV>	*Adverb*	*soudain, tout à coup*
<CONJC>	*Conjonction of coordination*	*et*
<CONJS>	*Conjonction of subordination*	*si, tant et si bien que*
<DET>	*Determiner*	*cette, la quasi-totalité de*
<INT>	*Interjection*	*aïe !, et merde !*
<N>	*Noun*	*pomme, pomme de terre*
<PREP>	*Preposition*	*de, à l'encontre de*
<PRO>	*Pronoun*	*me, quelqu'un*
<V>	*Verb*	*manger, s'entre-déchirer*
<X>	*Non-autonomous constituent of a compound word*	*aujourd, parce*

We can now be more precise: INTEX does not know what these codes signify, and it cannot verify that the codes used in a rational expression or a graph, as entered by the user, are correct. For example, nothing prohibits the user from writing the following expression:

```
<ADVERBE>* <ADJ> <SUBSTANTIF>
```

Of course, this expression will not identify any sequence in the texts because the INTEX dictionaries do not use these codes (for example, no word is tagged with the code "ADJ" in the DELAF dictionary). The advantage of this "freedom of expression" is that the user is free to invent and add any code in the dictionary: this code will be instantly re-useable in any rational expression or graph.

Similarly, flectional codes found in the DELAF dictionary of French are as follows:

Code	Meaning
m	*Masculine*
f	*Feminine*
s	*Singular*
p	*Plural*
1, 2, 3	*1st, 2nd & 3rd person*
P	*Indicative Present tense*
I	*Indicative Imperfect tense*
S	*Subjunctive Present tense*
Y	*Imperative Present tense*
C	*Conditional Present tense*
J	*Simple Past tense*
W	*Infinitive*
G	*Present Participle*
K	*Past Participle*
F	*Future*

Note that this convention, used by the RELEX community, is very practical for languages with a great deal of inflection: for example, we cannot have 5 lexical entries for verbal forms such as *aide* (1st or 3rd person singular, in the indicative present or subjunctive present, or 2nd person singular of the imperative). But nothing prohibits an end user to decide to represent the form *avions* in the following manner:

```
avions,avion.N+Conc:mp
avions,avoir.V+aux:I1p
avions,avoir.V+aux:S1p
```

or even:

```
avions,avion.N+Conc+Masc+Plur
avions,avoir.V+aux+Imp+1+Plur
avions,avoir.V+aux+Subj+1+Plur
```

Furthermore, if a user adds a specialized dictionary in which the vocabulary proper to chemistry would be entered as follows:

```
acide,acide.N+Chimie
```

```
acidifier,acide.V+Chimie
```

the expression `<N+Chimie> <V+Chimie>` could instantly be applied to texts to look for sequences made up of a noun followed by an adjective (should it be verb?); similarly, the symbol `<acide>` (to identify all words in the “chemistry” vocabulary, associated to the canonical form "*acide*"). For documentational research applications wherein even the distinction between nouns and verbs would be helpful, we could build "DELAF dictionaries" of the type :

```
acide,acide.CHIMIE
acidifier,acide.CHIMIE
```

The only constraint is that there must be exact correspondence between the codes found in the dictionaries and those used in the rational expressions and graphs.

Before using lexical symbols in the rational expressions or graphs, be sure that the codes that you apply to the text are actually present in the dictionaries!

In the French module, the DELAF dictionary is presented in two versions:

The **delafm.bin** dictionary contains all the inflected forms of French, which represents 675 249 different inflected forms (and approximately twice as many lexical entries). This version of the dictionary contains no syntactic or distributional information; the words are grouped in three levels: the **+z1** code corresponds to the basic vocabulary of French; **+z2** corresponds to standard vocabulary, and **+z3** corresponds to technical vocabulary. See also [Garrigues 1997] for a justification of this classification.

The **delaf1.bin** dictionary contains all the inflected forms of basic French (i.e. those forms associated with the code **+z1**). This represents 420 371 different inflected forms. This time, the verbal forms are associated with codes of syntax, the name of the lexicon-grammar table in which the verb is described, as well as some properties **+t** (transitive), **+p** (pronominal), **+i** (intransitive). The nouns are associated to a distributional code (**+Hum**, **+Conc**, etc.). Here are some syntactico-semantic codes present in the DELAF1 dictionary:

Code	Meaning
Abst	*(Noun) Abstract*
Anl	*(Noun) Animal*
AnlColl	*(Noun) Animal, Collective*
Conc	*(Noun) Concrete*
ConcColl	*(Noun) Concrete Collective*
Hum	*(Noun) Human*

HumColl	*(Noun) Human Collective*
t	*(Verb) transitive*
i	*(Verb) intransitive*
en	*(Verb) obligatory particle “en”*
se	*(Verb) reflexive*
ne	*(Verb) obligatory negation*
aux	*(Verb) auxiliary*
1, 2...	*(Verb) table 1, 2...*

Here are the first entries from the DELAF1 dictionary:

```
a,avoir.V+aux:P3s
a,avoir.V+en+i+35R:P3s
a,avoir.V+i+1:P3s
a,avoir.V+i+31R:P3s
a,avoir.V+i+35R:P3s
a,avoir.V+ne+i+1:P3s
a,avoir.V+ne+t+16:P3s
a,avoir.V+t+10:P3s
a,avoir.V+t+32H:P3s
a,avoir.V+t+32R3:P3s
a,avoir.V+t+36DT:P3s
a,avoir.V+t+37E:P3s
a,avoir.V+t+38L0:P3s
a,avoir.V+t+38LR:P3s
a,avoir.V+t+38R:P3s
a,avoir.V+t+39:P3s
a,avoir.V+t+6:P3s
a,avoir.V+t+U+32NM:P3s
a,.N:ms:mp
abaissa,abaisser.V+se+i:J3s
abaissa,abaisser.V+t+11:J3s
abaissa,abaisser.V+t+32RA:J3s
abaissa,abaisser.V+t+38L:J3s
```

The verb *avoir* has 18 syntactico-semantic uses in the lexicon-grammar; the verb *abaisser* has 4 syntactico-semantic uses. It is not necessary to qualify the fact that the DELAF1 dictionary is not really well suited to all current "simple" technical applications, and a good knowledge of linguistics is required to benefit from this information.

The DELAF dictionary of French has been conceived to identify *all* simple forms of common words in French.

INTEX contains a few other small DELAF format dictionaries, such as, for example, the dictionary of given names (roughly 500 entries); here are two such entries:

```
Abraham,.N+PR+Hum:ms
Dominique,.N+PR+Hum:ms:fs
```

When the lemma is identical to the DELAF entry, it is not useful to repeat it between the comma and the period. Further on we will discuss the standard format of the DELAF dictionaries, which must be respected.

Other, more specialized dictionaries are also available on the INTEX websites http://intex.univ-fcomte.fr : the dictionary **Prolintex** (put together by a team under the leadership of Denis Maurel) contains more than 73 000 forms of homonyms (*France, Paris*, etc.); Cédrick Fairon's dictionary of **nouns of professions** contains more than 4000 nouns. If you have put together such dictionaries, allow the entire INTEX community to benefit by sending us your files (with documentation!).

11.3. DELACF type electronic dictionaries

The dictionaries of compound words (DELACF) are similar to DELAF dictionaries of simple words; the difference is that the lexical entries (the text that appears at the beginning of the line and before the comma in the dictionary) and/or the lemmas (the text between the comma and the period) can contain separators. As an example, here are several entries from the French DELACF dictionary:

```
cousins germains,cousin germain.N+NA+Hum:mp
criant de vérité,.A+EPN:ms
pommes de terre,pomme de terre.N+NDN+Conc:fp
tant et si bien que,.CONJS+3
tout à coup,tout à coup.ADV+PCPC
```

In principle, the DELACF dictionaries are automatically built up from the DELAC dictionary; see [Silberztein 1993] and [Agata Chrobot, 2000] for two examples of implementation; Blandine Courtois (at LADL) and Cristina Mota (Univ. of Lisbon) also have programs that can generate DELACFs (unfortunately they have not been included in INTEX).

Free nominal groups vs. compound words

An important problem faced by the project of describing, on a wide scale, natural languages is that of the limit between compound words (that must be lexicalized) and free nominal groups. It is evident for those who perform automatic analyses on natural language texts, that in the following examples:

```
un cordon bleu, une corde bleue
```

The first nominal group must be lexicalized, but not the second. If *cordon bleu* is not included in a dictionary, a computer would neither be able to predict the meaning ("bon cuisinier") nor the associated lexical properties (for example, human noun), and certain applications, such as automatic translation would provide incorrect results.

Between these two extremes (a frozen compound noun and a free nominal group), there exist several hundred thousand more difficult cases, such as:

```
carte routière, ceinture noire, machine à laver
```

I adopted a set of criteria that would establish limits between those nominal groups that should be lexicalized and those that we choose not to lexicalize (cf. [Silberztein 1993]). The three primary criteria are:

Semantic atomicity: if the exact meaning of a nominal group cannot be deducted from the meaning of the components, the nominal group must be lexicalized (=> it is therefore treated as a compound noun).

For example, the noun *carte* possesses a dozen or so meanings (*carte géographique, carte de paiment, carte d'abonnement, carte d'identification, carte électronique, carte de visite*, etc.); only the first meaning of *carte* (= "map") is pertinent in the nominal group *carte routière,* which cannot mean: *carte de paiement* or *d'abonnement* for roads ("toll card"), or *carte d'identité* for those who use the roads ("road ID card"), or *carte électronique* used for truckers' necessities ("road map board"), etc.

The only way to account for the unexpected absence of ambiguity of the nominal group *carte routière* (contrary to the nominal groups *carte ancienne* or *carte plastifiée,* which are ambiguous), is to lexicalize it, which amounts to treating it as a compound noun.

Distributional restriction: if certain constituents of the nominal group, which by the way, belong to certain natural distributional classes, cannot be freely replaced, then we must acknowledge this distributional restriction by classifying the series of nominal groups in a lexicon, which again, amounts to treating it as a compound noun.

For example, only seven adjectives of color can combine with *ceinture* (belt) in a nominal group representative of a human noun: "*Luc est ceinture noire"* (Luc is black belt). These adjectives of color are not predictable a priori; they are not the same adjectives that we would find in "*Luc est maillot jaune"* ("*Luc is yellow jersey"*) or "*Luc est un col blanc*" ("*Luc is white collar"*).

Consequently, we must classify these seven nominal groups, which amounts to treating them as compound nouns.

Institutionalization of the usage: certain nominal groups, even those that are semantically and distributionally "free", are used in a quasi-obligatory manner, to the detriment of other potential syntactic constructions that are just as valid, but are never

used. One must understand that these nominal groups represent the "institutionalized" manner to refer to objects or concepts, and so to classify them, which also amounts to processing them like compound nouns.

For example, the French language would, a priori, allow us to call *machines à laver* (washing machines) by at least a dozen different names:

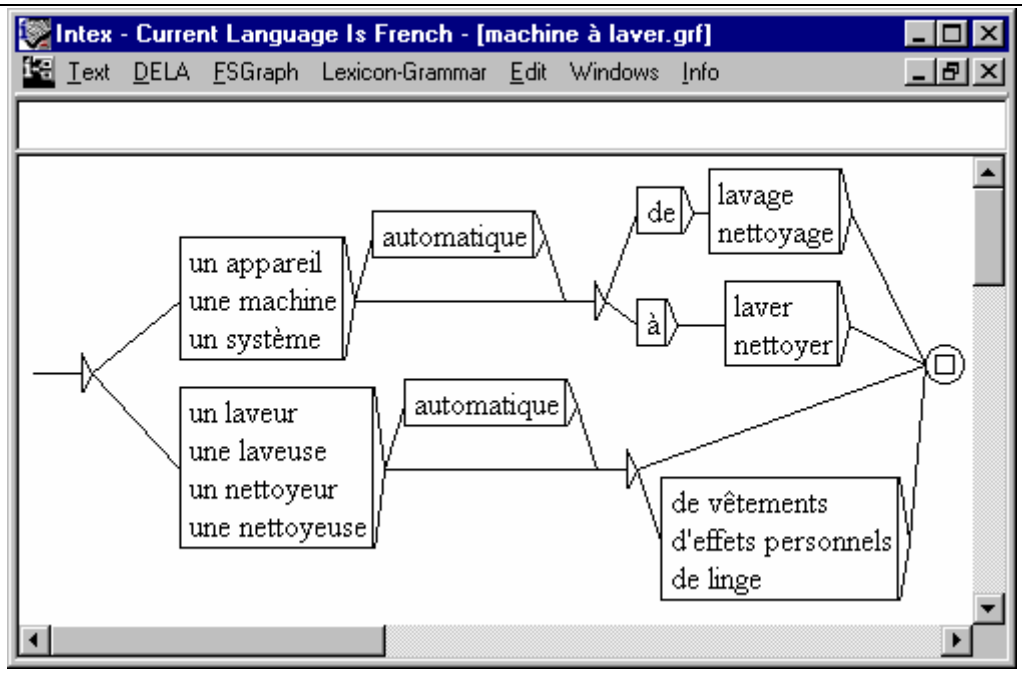


Figure 61. Several syntactically correct variants

It is remarkable that among all of these permitted nominal groups, only three are commonly used:

un lave-linge, une machine à laver, une machine à laver le linge

(French speakers in Québec use the term *laveuse*). It is essential to make special note of these three particular nominal groups in order to avoid possible mis-interpretations (*washing-machine => machine lavante*), to correctly index documents (for example *machine à laver* must be indexed with *lave-linge*, but not with *lave-vaisselle*), for pedagogical applications (French as second language teaching), and more generally to describe the vocabulary of the language: French speakers *know* these nominal groups; they are included in their vocabulary.

These three criteria define a group of compound words much larger than that which is generally assumed. Our estimations show that in order to cover standard French vocabulary, approximately 250,000 compound words must be classified and described.

Specialized uses of the DELACF dictionaries

The canonical form, which generally assumes the role of the "lemma", can be used to associate, with each compound form, a longer and more explicit variant; for example:

```
Etats-Unis,Etats-Unis d'Amérique.N+Géo+Pays:mp
U.S.A.,Etats-Unis d'Amérique.N+Géo+Pays:mp
USA,Etats-Unis d'Amérique.N+Géo+Pays:mp
```

Inversely, the author of a dictionary can choose a simple word to be the canonical form:

```
roman policier,polar.N+Conc:ms
roman policier de la série noire,polar.N+Conc:ms
```

The dictionaries of compound words can also contain entries which are in fact simple words, for example:

```
carte,carte bancaire.N+Conc:fs/carte bleue
carte,carte géographique.N+Conc:fs/carte routière
carte,carte électronique.N+Conc:fs/carte-mère
carte,carte postale.N+Conc:fs
```

This use of the DELACF dictionaries allows us to naturally understand numerous ambiguities associated with certain simple words; therefore, the simple word "*carte*" would be seen as ambiguous after consultation of the preceding dictionary.

The canonical form associated to each entry of the DELACF can also be used in the translation applications; for example:

```
carte bancaire,credit card.N+Conc:fs
carte bancaire,debit card.N+Conc:fs
carte routière,road map.N+Conc:fs
carte-mère,mother board.N+Conc:fs
carte postale,postcard.N+Conc:fs
```

Note that ambiguities in translation (*carte bancaire* = *credit card* or *debit card*) are handled by repeating the lexical entry.

The fact that simple words can be described in the DELACF format dictionaries allows us to fuse the both the DELAF and DELACF dictionaries, which could prove interesting for certain applications.

Caution: certain compound words cannot be entered into a DELACF dictionary purely for reasons of form:
-- compound words which begin with a non-alphabetical character (separator or number), for example "*2ème*" ;
-- compound words that contain a comma, for example: "*non, non et non !*".

Similarly, entries in the DELACF cannot be associated with a lemma that contains periods (eg. "*S.N.C.F.*"), since the period also serves to separate the "lemma" field from the "lexical information" field.

DELACF type dictionaries must be stored in the folder **Delacf** of the current language. The French DELACF dictionary is currently available in four files:

The file **Noms.bin** contains 244 726 forms of compound nouns, spread over a dozen or so classes, which correspond to the morpho-syntactic structure of the entries: for example, **+NA** for *Noun Adjective*, **+NDN** for *Noun 'de' Noun*, etc. Cf. B. Courtois, M. Silberztein Eds 1990 for a description of the types of compound nouns;

The file **Adv.bin** contains 7 753 frozen adverbs, either in a form that functions as an adverb (eg. *dans l'intimité la plus stricte*), or in a form that functions as a preposition (eg. *à grand renfort de)*; each entry is associated with the lexicon-grammar table in which the syntactic properties of the corresponding adverb are described (cf. M. Gross 1986);

The file **EPN.bin** contains 14 551 frozen forms used with the verb *être*, in forms that function as adjectives (eg. *criant de vérité*), adverbs (eg. *à un epsilon près*) or prepositions (eg. *une planche de salut pour*). Cf. M. Gross 1997 for a description of **EPN**s;

The file **Conjs.bin** contains 1 591 conjunctions of subordination, either in a form which functions as a conjunction (eg. *tant et si bien que*), an adverb (eg. *dans ce cas*) or a preposition (eg. *à défaut de*).

INTEX also contains, for French, some examples of compound word dictionaries, as for example a dictionary of pronouns (eg. *quelques-uns*), toponyms (eg. *New-York*, *Afrique du Sud*), first names (eg. *Jean-Paul*), etc.

Other dictionaries are available on the INTEX website: Pierre-André Buvet's dictionary of **nominal determiners** (*abondance, années lumière*, etc.) classifies more than 3000 compound determiners. If you have put together such dictionaries, allow the entire INTEX community to benefit by sending us your files (with documentation!).

11.4. Lexical Transducers

Lexical transducers are used to recognize forms of the text, and associate them with a lemma and some linguistic information, exactly in the same way as the DELAF and DELACF electronic dictionaries. Lexical transducers are also used to recognize and tag frozen expressions.

Lexical Transducers for Simple words

See next chapter (Tokenization & Morphology).

An example of lexical transducers for compound words

Dnum composés-.fst: this lexical transducer, stored in the **Delacf** file, identifies compound numerical determiners (eg. "*cent trente-trois mille deux cents*"), and associates linguistic information to them ("*DET:mp:fp*"). This transducer is built from a series of 9 graphs, of which two can be seen below:

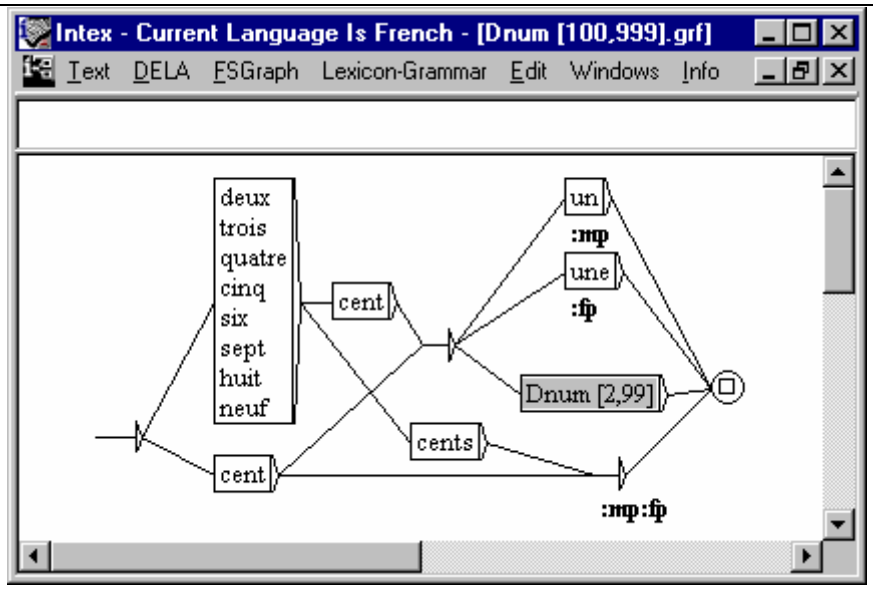


Figure 62. Numerical Determiners from 100 to 999

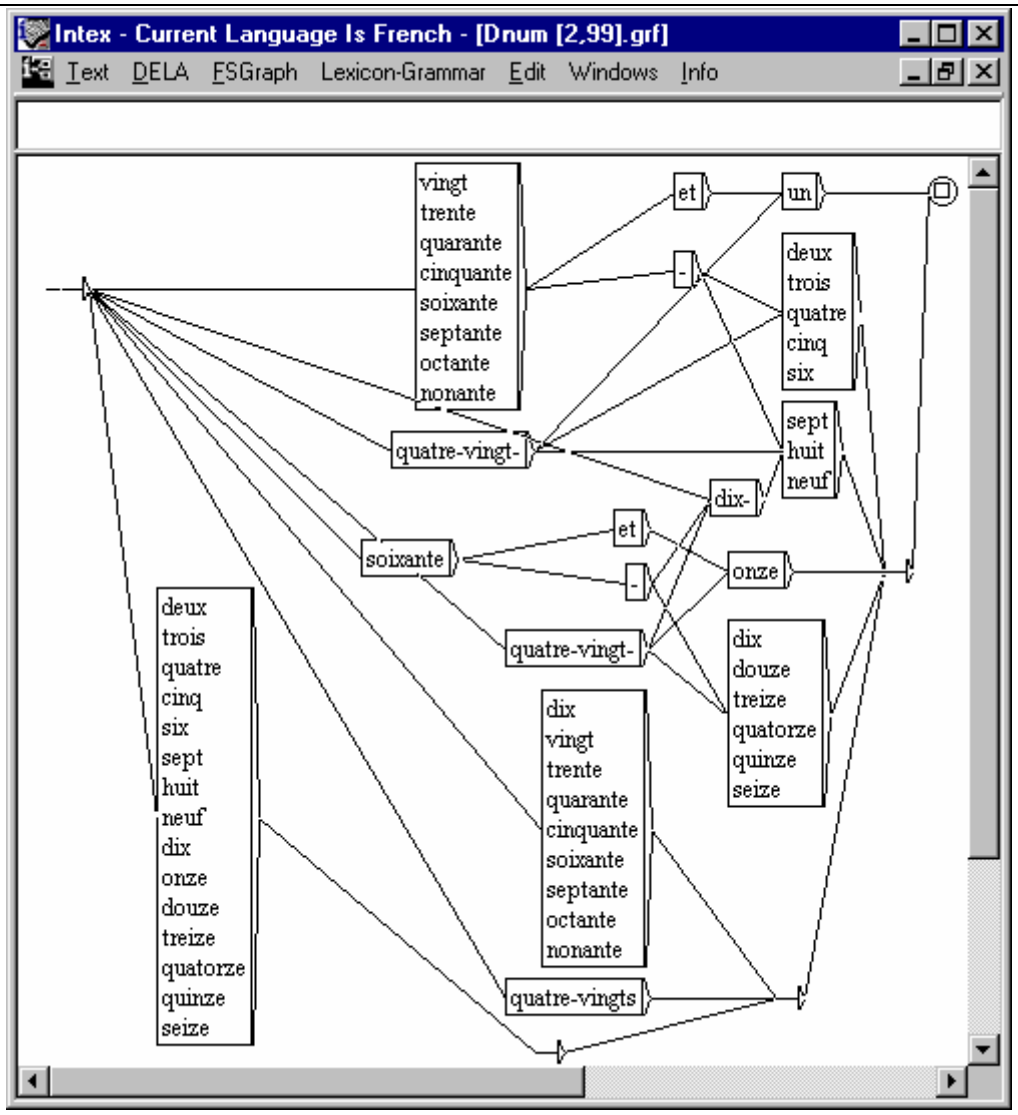


Figure 63. Determiners from 2 to 99

Even the subtleties of French spelling have been taken into account (*deux cent**s** tables* ; *deux cent trois tables*), as well as national variants (for example, the transducer will recognize both *septante* and *soixante-dix*).

An example of a transducer for lexical frozen expressions

Perdre la raison.fst: this lexical transducer, stored in the **Delae** folder, recognizes variants of this expression, that could potentially contain insertions (eg. "*Luc a* perdu *soudain* les pédales"):

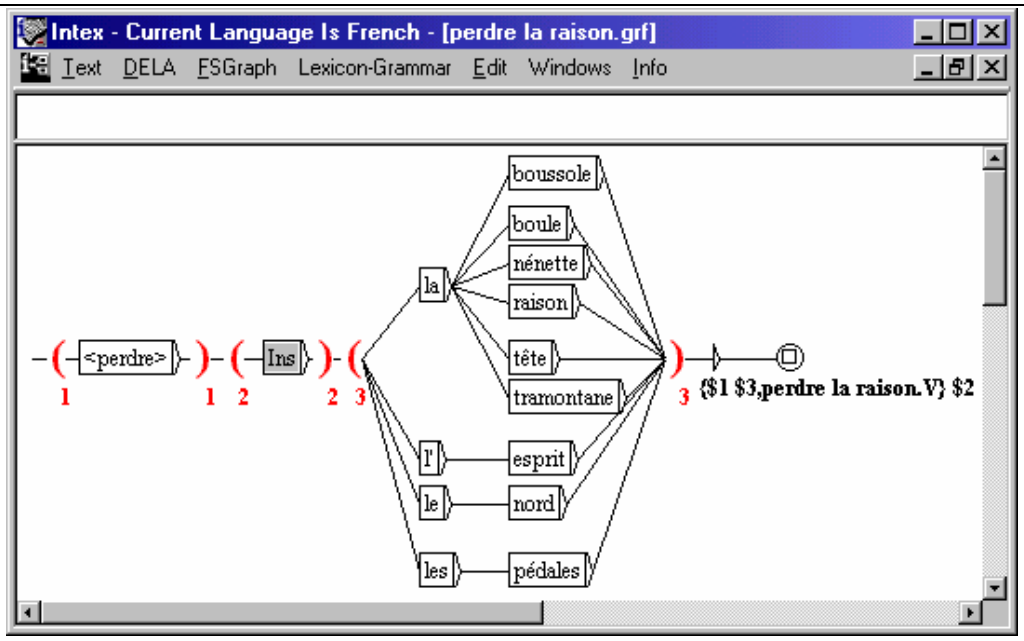


Figure 64. A frozen expression and its variants

The transducers of rational expressions are improved transducers, because there is a need to keep in mind the insertions that can appear between constituents of the expressions and highlight them within the lexical entries (remembering that this characteristic, by definition, distinguishes frozen expressions from compound words.).

For example, if the preceding transducer is applied to the text: "*Luc a perdu soudain les pédales"*, the form "*perdu"* will be placed in the variable **$1**, the form "*soudain"* in the variable **$2**, and the sequence "*les pédales"* in the variable **$3**. The result would therefore be:

```
Luc a {perdu les pédales,perdre la raison.V} soudain
```

This transducer issues forth from a graph drawn up manually. We will see further on that we can build such automatic transducers from lexicon-grammar tables.

Processing embedded graphs in the transducers of frozen expressions

After having drawn a graph for frozen expressions (as in the preceding case), we must compile a corresponding transducer and store it in the "**Delae**" folder of the active language. Graphs called on by the primary graph (eg. Ins) are taken into account during this compilation; in general, these reference graphs are stored in the "**Graphs\Lib**" folder.

Meanwhile, the graphs embedded in the master graph used to build a transducer from a lexicon-grammar table are not taken into account when the transducer is compiled. They are taken into account during the search for frozen expression in the text (when

we apply the lexical resources to the text). Thanks to this function, we are not required to recompile the transducers each time we modify the reference graphs.

When we apply lexical resources for frozen expression, the names of the loaded reference graphs, as well as those not found by INTEX, are displayed in (**Info>Console**, or **F2**). For example, if we apply the transducer "**C1d"** to the text "**La femme de trente ans"**, theconsole will display:

```
> Grammar "Nmaladie" not found
> Loading "c:\Program files\Intex\French\Graphs\Lib\Dét.fst"
> Loading "c:\Program files\Intex\French\Graphs\Lib\LE.fst"
> Loading "c:\Mon Intex\French\Graphs\Lib\Ins.grf"
> Loading "c:\Program files\Intex\French\Graphs\Lib\se.fst"
> Grammar "Dnum-ième" not found
> Grammar "Saint-Nprénom" not found
> Grammar "Ngibier" not found
> Grammar "Ntps" not found
> Loading "c:\Program files\Intex\French\Graphs\Lib\Poss-0.fst"
> Grammar "Nuple" not found
> Loading "c:\Program files\Intex\French\Graphs\Lib\que.fst"
```

It is of course important to keep these graphs up to date. The graphs Nmaladie, Dnum-ième, Saint-Prénom, Ngibier, Ntps and Nuple do not yet exist: consider this a call for volunteers!

Abbreviations

Certain frozen expressions from table **C1d** accept abbreviation (property **N0 V**); for example:

abandonner => abandonner la partie
boire => boire le coup
courir => courir la gueuse, courir la prétentaine, courir le cotillon, courir le guilledou, courir les filles, courir les garçons, courir les honneurs, courir les jupons

Consequently, numerous simple words will (for the most part, incorrectly) be associated with one or more frozen expressions by the lexical module. For example, consider the two sentences, in which the expression *boire un coup* was identified:

```
Luc a bu de l'eau. Luc boit souvent dans ce café.
```

In the first, the syntactic analyzer should have completely eliminated the expression since the direct object *de l'eau* is explicit. On the other hand, the expression remains active in the second sentence.

11.5. Classification of priorities

The users can select any number of dictionaries and lexical transducers for simple words, compound words or frozen expressions and apply them to a text in order to extract the vocabulary. It is easy to *add* information: it merely requires creating a dictionary or a transducer, placing it in the correct folder (**Delaf**, **Delacf** or **Delae**) and selecting it along with the other lexical resources. One can also *hide* information in INTEX thanks to a system of prioritization. Each lexical resource is linked to one of three priority levels, used during the consultation:

(1) "Priority" resources are applied to the text first;

(2) When a form is not recognized, in other words, when the application of priority lexical resources has not found anything (and only in this case), INTEX applies the "normale" priority lexical resources;

(3) when the consultation of "priority" and "normal" lexical resources has found nothing, INTEX applies the "minimal" priority lexical resources.

This system allows users to hide or add linguistic information at will.

For example, the French dictionary **Delafm** (stored in the "`Delafm.bin`" folder) has a "normal" priority. It describes numerous usages that are not overly frequent, for example *la*, *si* = masculine nouns (musical notes – *la & ti in English*), *a* = masculine noun (name of the first letter of the alphabet), *par* = masculine noun (golf term), etc. For a specific application, as in the processing documents of the technical domain, in which these words would never appear, it is useful to create a smaller, prioritized dictionary with respect to the **Delafm** dictionary, in which these uses are not described: this smaller dictionary allows us to hide useless entries from the **Delafm**. Consider the dictionary "`Filtre-.dic`":

```
a,avoir.V:P3s
la,le.DET:fs
la,le.PRO:fs
par,par.PREP
si,si.CONJS
```

The primary dictionaries and transducers have a name that ends with the character "`-`". If the dictionaries "`Delafm.bin`" and "`Filtre-.dic`" are applied together, the forms *a*, *la*, *par* and *si* will be associated only with the uses mentioned above (for these forms, INTEX ignores the dictionary **Delafm**); for all other forms, , INTEX will consult the **Delafm** dictionary.

The "minimal" priority dictionaries & transducers are applied when application of the other lexical resources has failed. Dictionaries with minimal priority must have names that end with the character "+".

We can use this mechanism to process unknown words (cf. the next chapter; example of the graph **Npropre+.grf** , describing proper names). All simple forms not found in the INTEX dictionaries and which begin with a capital letter, are identified by this transducer. Similar transducers can be used to recognize productive morphological derivations such as *déstalinisation*, *jospinisme*, *balladurette*, *fabiusien*, etc.

For the dictionaries and transducers of compound words, the importance of a maximum priority is slightly different, even if their operation is the same:

-- if a lexical resource of compound words takes precedence over lexical resources of simple words, the compound words found will not be further deconstructed by INTEX (otherwise, compound words would, a priori, always be considered ambiguous with sequences of simple forms). For example, if the following form is an entry in the prioritized dictionary of compound nouns:

```
sous-marin nucléaire d’attaque,.N+Conc+Milit:ms
```

the following sequence will be treated as a single, non-ambiguous word:

…sous-marin nucléaire d’attaque…

INTEX will ignore the simple words *sous*, *marin*, *nucléaire*, *d* and *attaque*.

-- if more than one form of compound words of varying lengths are identified, by a prioritized lexical resource of compound words, as being in the same position, then only the longest forms will be taken into consideration by the system. For example, if the two following forms are entries in a prioritized dictionary:

```
dans l’intimité,.ADV+PDETC
dans l’intimité la plus stricte,.ADV+PCDC
```

the following sequence will not be treated as ambiguous:

…dans l’intimité la plus stricte…

as, at first it could’ve been considered ambiguous: either the adverb **PDET** followed by the three simple words *la*, *plus* and *stricte*, or the adverb **PCDC**.

11.6. The vocabulary of the text

The application to a text, of lexical resources for simple words, compound words and frozen expressions (**Apply lexical resources**) produces the *vocabulary of the text*, or the full inventory of linguistic units, from the selected lexical resources, found in the text. The vocabulary of the text is stored in the folder associated with the text (if the text file is "**toto.txt"**, the folder of the text is "**toto_txt"**).

This vocabulary (see below) represents four files:

-- the **DLF** file contains all the simple forms of the text, associated with the lexical information found in the selected lexical resources of simple words.;

-- the **ERR** file contains all the simple forms from the text that were not found in the selected lexical resources of simple words;

-- the **DLC** file contains all the sequences of simple forms, from the text, that were found in the selected lexical resources for compound words;

-- the **DLE** file contains all the sequences of simple forms, from the text, that were found in the selected lexical resources for frozen expressions.

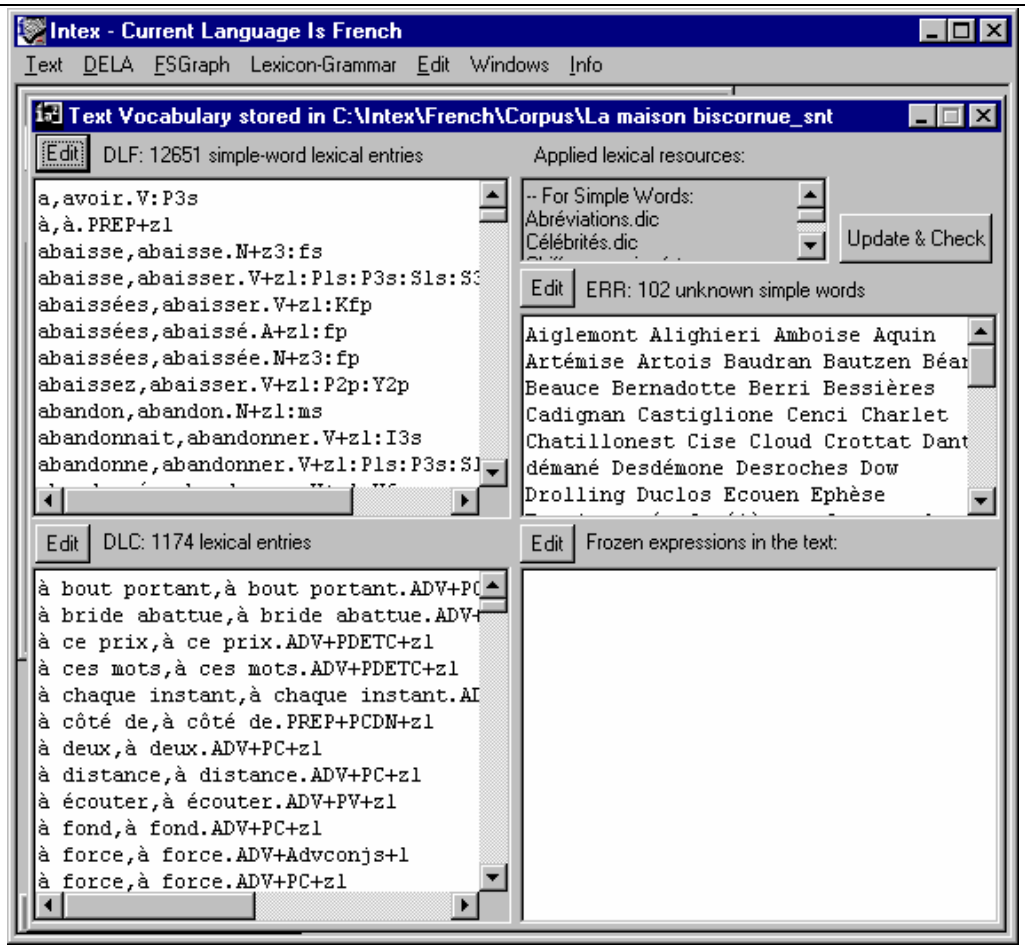


Figure 65. Vocabulary of the text

Caution: it is not simply because a form (or sequence of forms) was found in a lexical resource, that the linguistic unit corresponding to the lexical entry actually appears in the text.

For example, if we apply the lexical resources DELAFM and DELACF to the following text:

```
Luc en fait le tour
```

The preposition *en*, the adverb *en fait* and the preverbal pronoun *le* will appear in the vocabulary of the text, even if these linguistic units do not appear in the text. Next, you'd have to apply certain methods to remove all ambiguity in order to eliminate incorrect lexical hypotheses.

The dialogue box "Applied lexical resources" shows the names of lexical resources that were applied to obtain the vocabulary showing. The vocabulary contains the lexical resources that will be used by the grammatical units of INTEX: **Locate pattern** and **Parse**.

It is often interesting to edit the four files already presented, by hand, in order to more precisely adjust the vocabulary to the text, e.g. to eliminate non-pertinent ambiguities

in the text, or to describe the words not taken into inventory by the general lexical resources. Note that in this case, the "Applied lexical resources" zone in the dialogue box will mention these modifications.

Chapter 12. TOKENIZATION & MORPHOLOGY

Word forms are generally recognized by a simple lookup of a DELAF-type dictionary, in which they are explicitly associated with a lemma and some linguistic information. But there are cases where it is easier, and more natural, to use morphological rules, rather than dictionaries, to represent word forms. In INTEX, morphological rules are implemented in the form of “lexical transducers”, that are graphs which input is used to recognize word forms, and which output is used to compute the corresponding lemma and lexical information. Lexical transducers, just like dictionaries, are stored in the Delaf directory of the current language, and the results of the morphological parsing are stored in the vocabulary of the text, exactly as for dictionaries.

A lexical transducer (i.e. a morphological rule) can be as simple as a graph that recognizes a fixed set of word forms, and associates them with some linguistic information. More complex rules can recursively split word forms into smaller parts (affixes) that are stored in variables, and then enforce simple or more complex morpho-syntactic constraints to each variable.

12.1. Transducers with no lexical constraints

Generally, one uses lexical transducers when the number of word forms to be represented is large, whereas these forms are easily produced by a few productive rules.

An orthographical transducer

Here is an example of an elementary transducer that recognizes a few spelling variants:

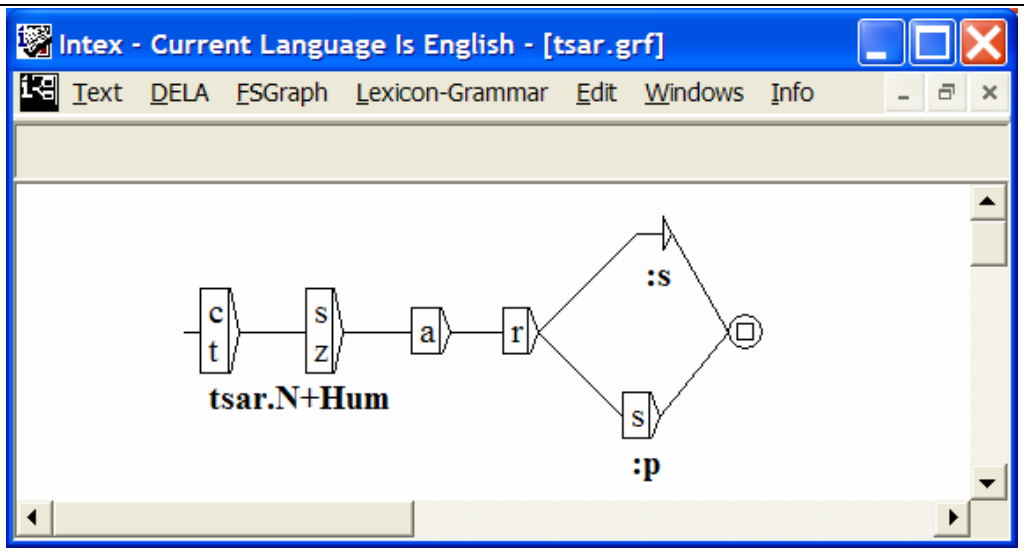


*Figure 66. Graph **tsar***

This graph recognizes the four word forms *csar*, *czar*, *tsar*, *tzar* and associates them with the lemma *tsar* and the information "**N+Hum:s**" (*Noun*, *Human*, *singular*); it also recognizes the four word forms *csars*, *czars*, *tsars*, *tzars* and associates them with the same lemma *tsar*, and with the information "**N+Hum:p**" (*Noun*, *Human*, *plural*).

This graph is equivalent to the following DELAF-type dictionary:

```
csar,tsar.N+Hum:s
csars,tsar.N+Hum:p
czar,tsar.N+Hum:s
czars,tsar.N+Hum:p
tsar,tsar.N+Hum:s
tsars,tsar.N+Hum:p
tzar,tsar.N+Hum:s
tzars,tsar.N+Hum:p
```

This graph, and similar ones, can be used by NLP applications to associate "variants" of a word or a term (whether orthographic, phonetic, synonymous, semantic,

translations, etc.) with one particular canonical form, that acts as an index key, an hyperonym, or a “super lemma”. This feature allows extractors or search engines to index and retrieve hyponyms or variants of a given term (even if these variants are compounds), e.g. the query “germ” would link to “variants” such as *bacteries*, *decease*, *sickness*, *virus*, etc.

A simple transducer for unknown words

Another example of an elementary lexical transducer is the proper name recognizer:

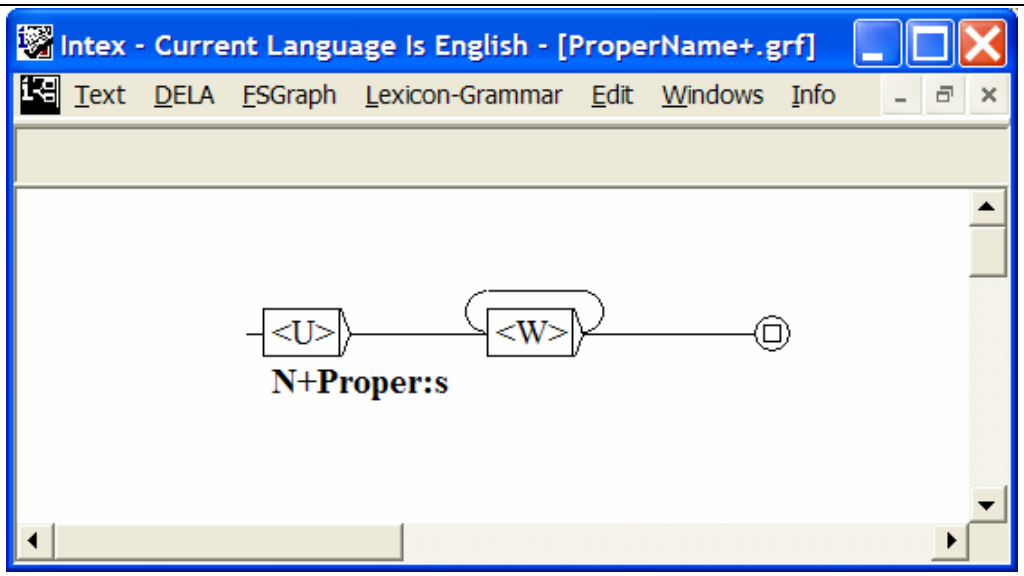


Figure 67. Recognition of proper names

The symbol <U> matches any uppercase letter; the symbol <W> matches any lowercase letter. Hence, this graph recognizes all word forms that begin with an uppercase letter, followed by one or more lowercase letters. For instance, the graph matches the word forms “Ab” and “John”, but not “abc”, “A” or “INTRODUCTION”. All recognized word forms are associated with the corresponding linguistic information: **N+Proper:s** (*Noun*, *Proper Name*, *singular*).

One difference with the previous graph **tsar**: since the transducer **ProperNames+** does not produce any lemma (just the information), each recognized word form will be considered as its own lemma. For instance the word forms “John” and “Mary” will be recognized as if the following entry:

```
John,John.N+Proper:s
Mary,Mary.N+Proper:s
```

were actually listed in a DELAF-type dictionary.

Usually, one gives this types of generic graphs a low priority, so that only word forms that were **not** recognized by the lookup of the other dictionaries are processed (the

character "+" at the end of the file name **ProperName+.grf** tells INTEX to apply this lexical resource **after** all the others).

INTEX's morphological module uses the following special symbols:

<L>	*any* **L***etter*
<U>	*any* **U***ppercase letter*
<W>	*any lo***W***ercase letter*
<A>	*any* **A***ccented letter*
<N>	*any u***N***accented letter*

Computing lexical information

It is possible to design transducers that compute information during the recognition process. For instance, consider the following transducer **RomanNumerals10-99**:

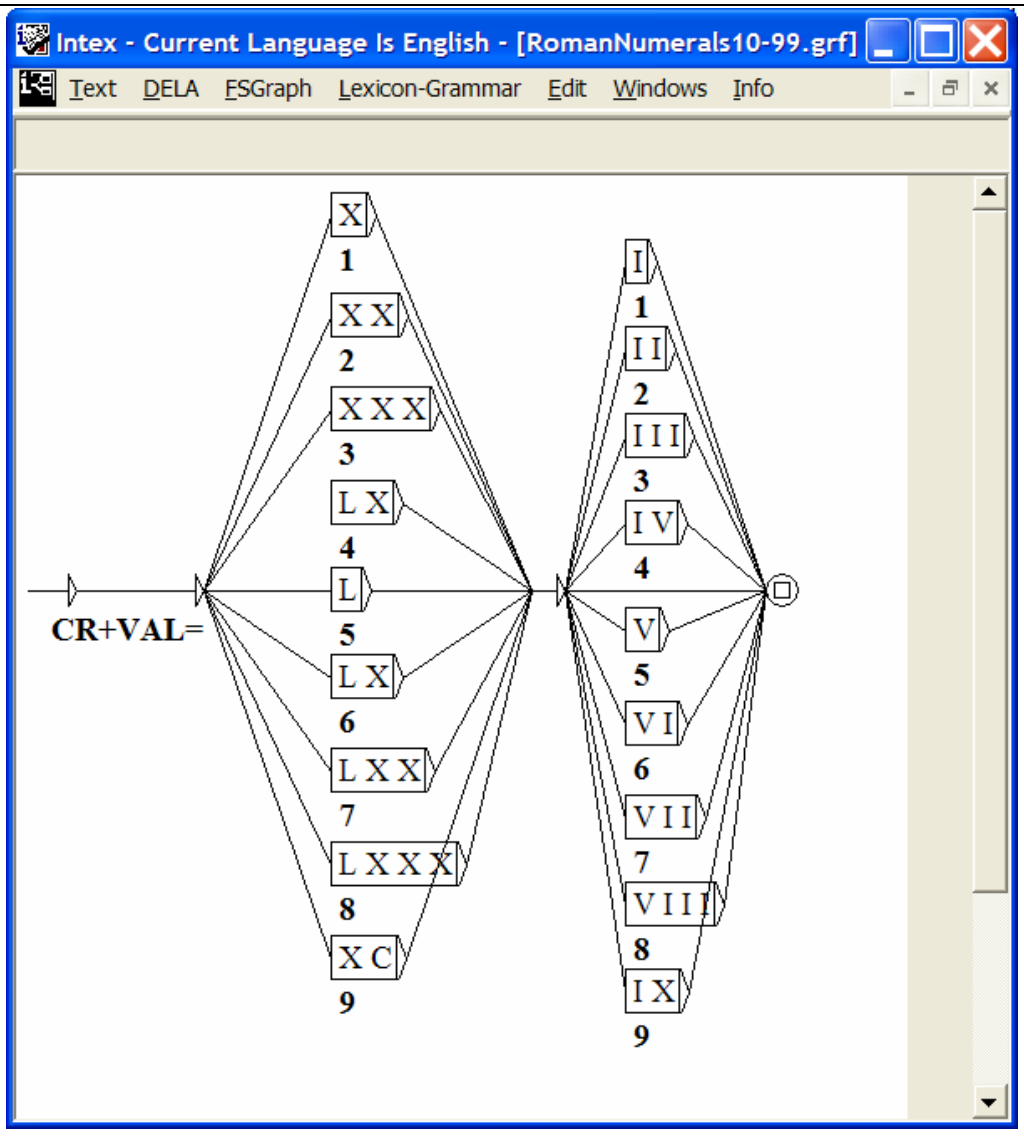


Figure 68. A graph for roman numerals

Thanks to this graph, word forms such as “LIV” and “XXXII” will be processed exactly as if they were actual entries of a DELAF-type dictionary, i.e.:

```
LIV,LIV.CR+VAL=54
XXXII,XXXII.CR+VAL=32
```

The transducer actually used by INTEX to recognize all the roman numerals from “I” to “MMMMM” is made of six graphs, and is stored in the Delaf directory.

Computing the lemma

In all the previous examples, the lemma associated with the word forms to be analyzed was either explicitly given (e.g. “tsar”), or implicitly identical to the word forms.

It is also possible to compute the lemma with lexical transducers. To do that, one must explicitly produce the actual tag that represents the linguistic unit (in INTEX, tags are lexical entries that are represented between “{“ and “}”). The explicit notation allows then to use INTEX enhanced transducers, i.e. graphs with variables (cf. Chapter 8. Transducers with variables). For instance, consider the following graph:

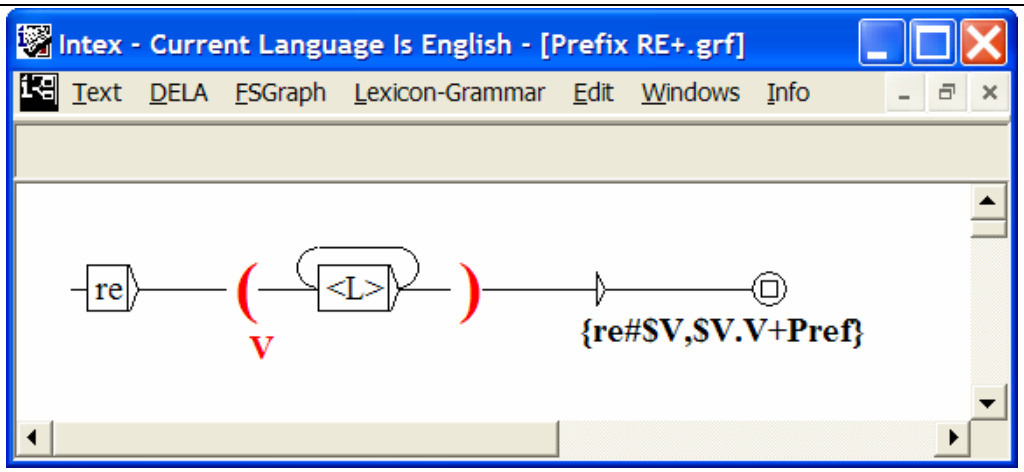


Figure 69. Removing the prefix “re-”

This graph recognizes any word form that starts with the prefix “re” (<L> stands for any letter). In the process, the suffix (i.e. the sequence of letters after “re”) is stored in the variable **$V**. Recognized word forms are then be associated with the corresponding tag:

```
{re#$V,$V.V+Pref}
```

in which all occurrences of the variable **$V** are replaced with its value. For instance, when the word form “repay” is recognized by the transducer, its suffix “pay” is stored in variable **$V**. The word form is then tagged as: `{repay,pay.V+Pref}`, that says that it is associated with the lemma “pay”, the category “V” (Verb) and the syntactic

feature “+Pref” (Prefixed Verb). Similarly, the word form “redo” will be tagged as `{redo,do.V+Pref}` (“redo” is associated with the lemma “do”; it is a prefixed Verb).

The following graph:

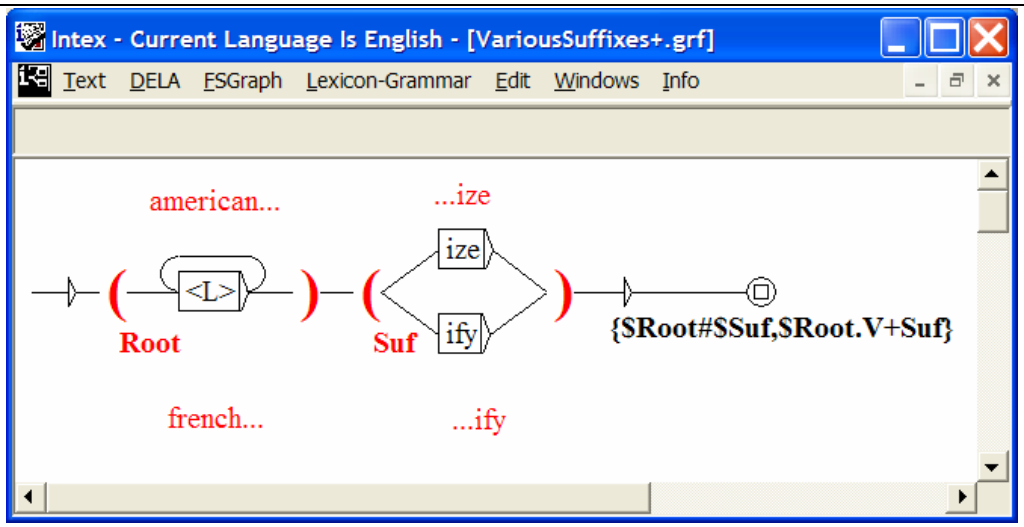
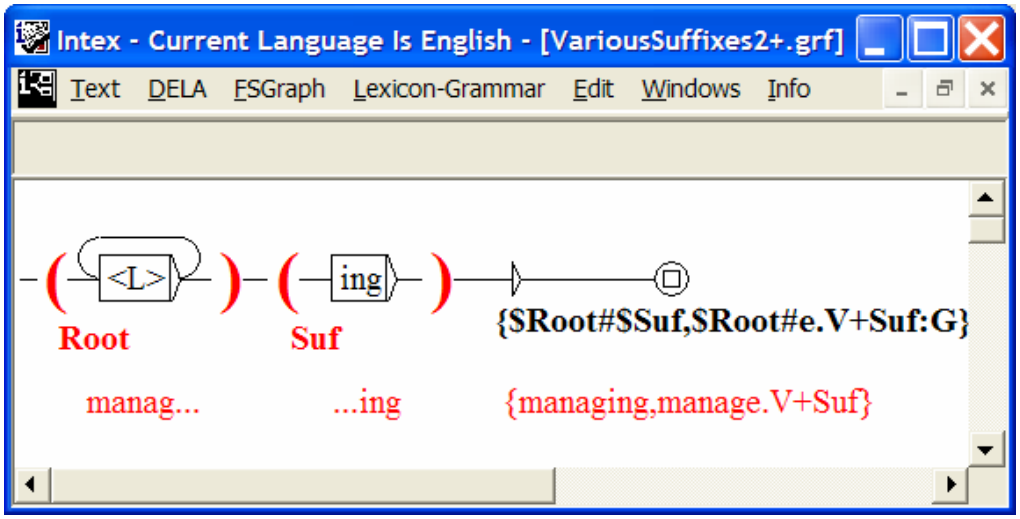


Figure 70. Removing various suffixes

will tag the word form “frenchify” as: `{frenchify,french.V+Suf}`, the word form “americanize” as: `{americanize,american.V+Suf}`.

By storing affixes of the word form into several variables, it is possible to process more complex morphological phenomena, such as the deletion or addition of letters at the boundaries between the root, prefixes and suffixes. For instance, the following graph can be used to process some particular English verbs that end with an “e” which is deleted for several forms (e.g. “manage” gives “managing”):



When the word form “managing” is recognized by the transducer; $Root stores the prefix “manag”, and $Suf stores the suffix “ing”. The resulting tag includes the entry

“$Root#$Suf” (i.e. “managing”), the resulting lemma “$Root#e” (i.e. “manage”) and the resulting information (V+Suf:G).

One word form represents more than one linguistic unit

Morphological transducers can also produce more than one linguistic unit (tag) for a particular word form. This feature allows one to process contracted words, such as the word form “*cannot*”; a transducer could then associate this single word form with a sequence of two tags:

```
{can,can.V} {not,not.ADV+Neg}
```

We will see below that the ability to produce more than one tag for a single word form is essential to the analysis of Asian and Germanic languages. This feature can also be used to replace simple word forms with complex expressions, e.g. “nicely” could be tagged as “`{in,in.PREP} {a,a.DET:s} {nice,nice.A} {way,way.N}`”.

Transducers with lexical constraints

The previous transducers can be used successfully when one can describe the exact set of word forms to be recognized (such as the roman numerals), or when the set of word forms is extremely productive (such as the proper names).

Indeed, the “generic” type of transducers that includes symbols such as <L> has proven to be useful in order to quickly populate dictionaries, or to automatically extract from large texts lists of word forms that will later be studied and manually checked by lexicographers. For instance, the patterns “<L>* ize” and “re <L>*” have been applied to large texts in order to enrich the DELAF dictionary.

When using these generic graphs, one must give them the lowest priority (i.e. add a “+” to the end of the file name), so that word forms that are already described in dictionaries are not being re-analyzed; for instance, in the previous graph, one does not want the word form “ping” to be re-analyzed as the verb “to pe”, in the gerundive!

It is also possible to apply lexical constraints to various parts of the word form. In INTEX, these constraints are produced by the morphological transducer, and are written between angles (“<” and “>”) (these are the same constraints that are used by the INTEX syntactic parser). For instance, the previous graph could be rewritten as:

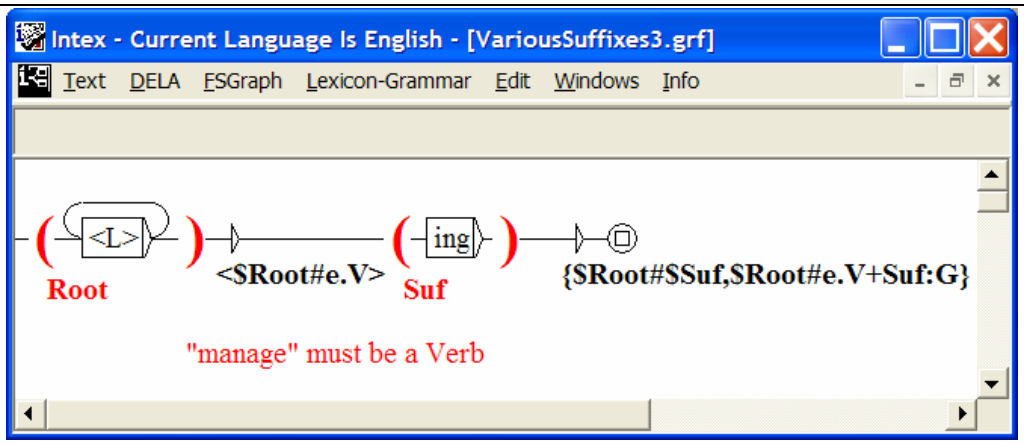


Figure 71. adding a lexical constraint

Just like the previous graph, this new graph does recognize both word forms “managing” and “ping”; however, the lexical constraint **<$Root#e.V>** gets rewritten as **<manage.V>**; this constraint is validated by a dictionary lookup, while the constraint **<pe.V>** does not since there is no entry “pe” in the DELAF dictionary that is associated with the code “V”. Therefore, only the first analysis is proposed.

Lexical constraints can be as complex as necessary; they constitute an important feature of INTEX because they give the morphological module unlimited precision, up to the precision of a dictionary. For instance, one can systematically check all the verbs of a dictionary that derive to an “-able” adjective, and associate them with the morpho-syntactic feature “**+able**”; for instance:

```
show,.V+able
laugh,.V+able
```

Then, the following graph:

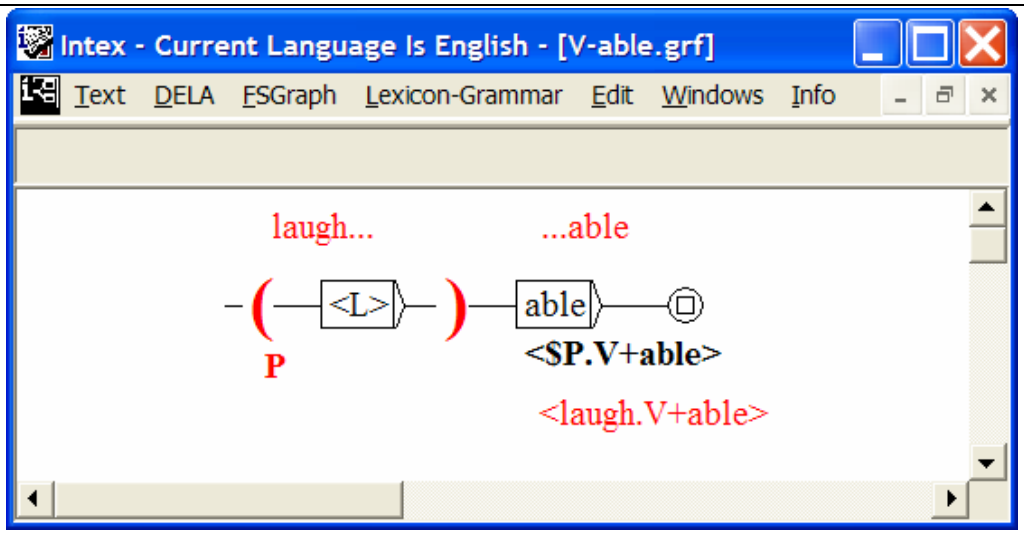


Figure 72. A simple derivation

ensures that derivations are performed only for verbs associated with the feature “+able”. The word forms “showable” and “laughable” are recognized because the

lexical entries “show” and “laugh” are associated with the feature “+able” in the dictionary; on the other hand, the word form “sleepable” and “smileable” are not recognized because the lexical entries “sleep” and “smile” would not be associated with the feature “+able”. Other word forms, such as “table”, would not be recognized because they do not derive from verbs (“t” is not a verb, therefore it does not meet the lexical constraint “V+able”).

Furthermore, it is possible to limit certain derivations to specific classes of words, e.g. only transitive verbs, only human nouns, or only adjectives of a certain distributional class. For instance, the graph:

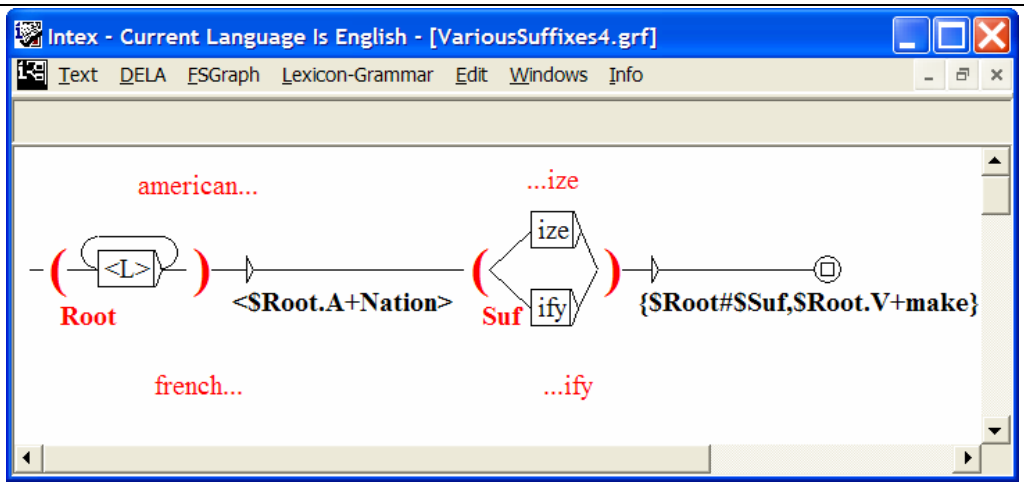


Figure 73. derivation with constraint

performs derivations only on adjectives that are associated with the code “+Nation”, such as “american” and “french”, and produce tags such as:

```
{americanize,american.V+make}
{frenchify,french.V+make}
```

It is also possible for a transducer to produce more than one lexical constraint, so that for instance a particular word form (e.g. “reaccountability”) is associated with a sequence of constraints such as “<$Pref.PREF+V> <$Root.V+t>” (“re” must be a prefix compatible with verbs, and “account” must correspond to a transitive verb).

Complex tokenizations

Tokenizing a word form into a series of linguistic units (i.e. tags) is essential for Asian and Germanic languages. For instance, the German word form:

Schiffahrtsgesellschaft

could be associated with three tags, rather than one:

```
{Schiff,schiff.N} {fahrt,fahren.V:K}
{gesellschaft,gesellschaft.N}
```

This complex tokenization can be performed by the following specific graph:

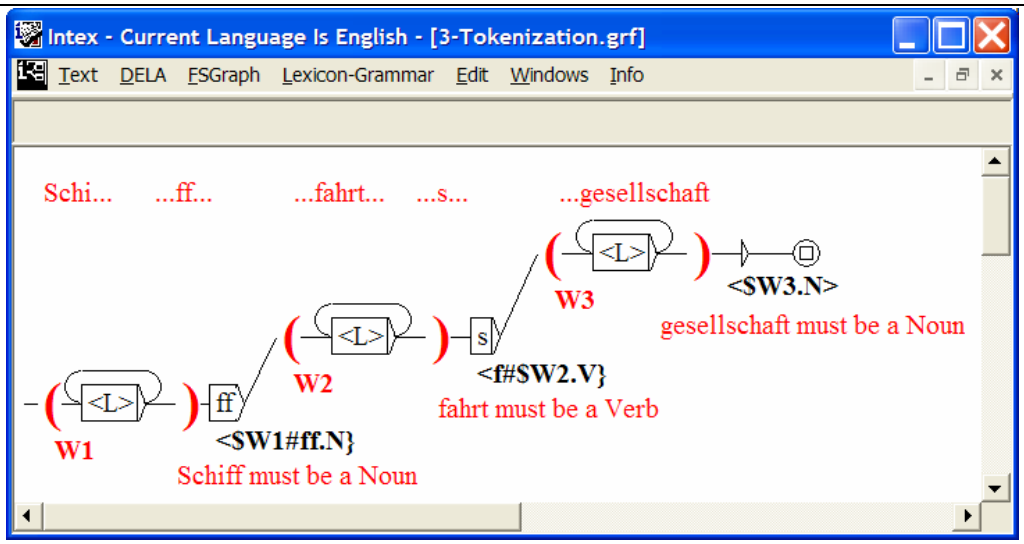


Figure 74. A complex tokenization in German

Note how the third (missing) “f” between “Schiff” and “fahren” is re-introduced in the resulting tags, and how the extra “s” between “fahrt” and “gesellschaft” is deleted.

Transfer of features and properties

INTEX enforces lexical constraints to affixes of the word form by looking up dictionaries. There, these affixes are associated with features and properties that can in turn be transferred to the resulting tag(s).

For instance, the word form “reestablished” can be linked to the verbal form “established”. In the dictionary, the verbal form “established” is itself associated with linguistic information, such as “Lemma = establish”, “+t” (transitive), “:I” (Preterit), etc.

These properties and features can then be transferred to the tag produced by the transducer, so that the resulting tag for “reestablished” inherits some of the properties of the verbal form “established”. INTEX uses special variables to store the value of the fields of the lexical information associate with each constraint. Lexical constraints (and their variables) are numbered from left to right ($1 being the first lexical constraint produced by the transducer; $2 the second, etc.), and the various fields of the lexicon are named “E” (**E**ntry of the dictionary), “L” (corresponding **L**emma), “C” (morpho-syntactic **C**ategory), “S” (**S**yntactic or semantic features) and “F” (in**F**lectional information). For instance:

$1E = 1st constraint, corresponding lexicon **E**ntry
$1L = 1st constraint, **L**emma
$1C = 1st constraint, **C**ategory
$1S = 1st constraint, **S**yntactic features

$1F = 1st constraint, in**F**lectional features

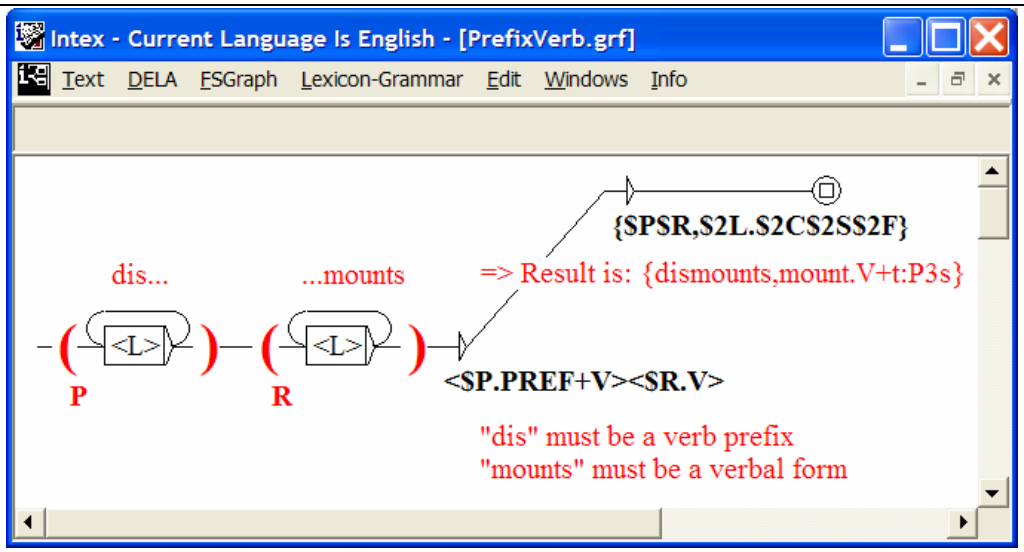


Figure 75. prefixes and verbs

If the word form “dismounts” is parsed by the previous graph, the resulting tag will be:

```
{$P$R,$2L.$2C$2S$2F}
```

In this tag, the variable **$P** stores the form “dis”; variable **$R** stores “mounts”; **$2L** stores the lemma of “mounts” in the dictionary, i.e. “mount”; **$2C** stores the category of “mounts”, i.e. “V” (verb); **$2S** stores the features for “mounts”, i.e. “+t” (transitive), and **$2F** stores the inflectional information for “mounts”, i.e. “:P3s” (Present, third person). As a result, the word form is tagged as:

```
{dismounts,mount.V+t:P3s}
```

Inflectional analysis without DELAFs

INTEX includes an inflectional module that automatically constructs DELAF-type (i.e. dictionaries in which all word forms are associated with their lemma) from DELAS-type dictionaries (i.e. only lemmas are stored). For instance, from a DELAS-type dictionary such as:

```
abandon,V3
help,V3
love,V3
```

Given the inflectional graph **V3**:

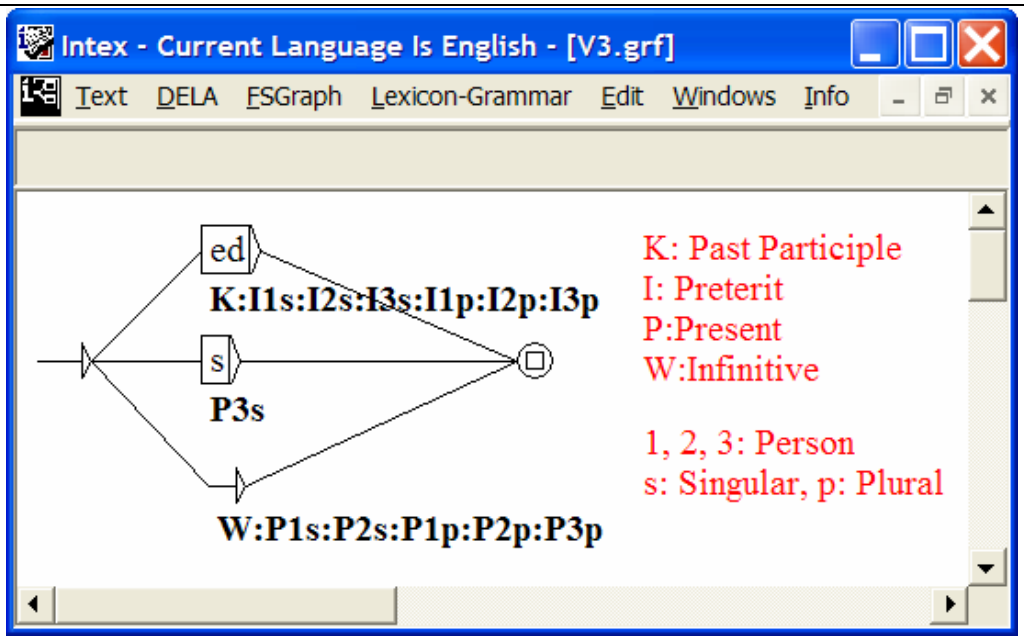


Figure 76. Inflectional transducer

INTEX automatically builds the following DELAF-type dictionary:

```
abandon,abandon.V:W:P1s:P2s:P1p:P2p:P3p
abandoned,abandon.V:K:I1s:I2s:I3s:I1p:I2p:I3p
abandons,abandon.V:P3s
help,help.V:W:P1s:P2s:P1p:P2p:P3p
helped,help.V:K:I1s:I2s:I3s:I1p:I2p:I3p
helps,help.V:P3s
love,love.V:W:P1s:P2s:P1p:P2p:P3p
loved,love.V:K:I1s:I2s:I3s:I1p:I2p:I3p
loves,love.V:P3s
```

This DELAF dictionary is in turn compressed into a Minimal Finite-State Automaton associated with a linear-time lookup procedure; lexical analysis of texts is then performed by a mere lookup of DELAF dictionaries.

For certain languages however, such as Hungarian or Korean, it is not possible to generate DELAF-type dictionaries, because the number of forms would be too large. In these cases, it is better to use a DELAS-type dictionary in conjunction with a morphological parser. For instance, here would be a dictionary:

```
abandon,.V+Conj3
help,.V+Conj3
love,.V+Conj3
```

Then, the morphological conjugation graph **Conj3** would be:

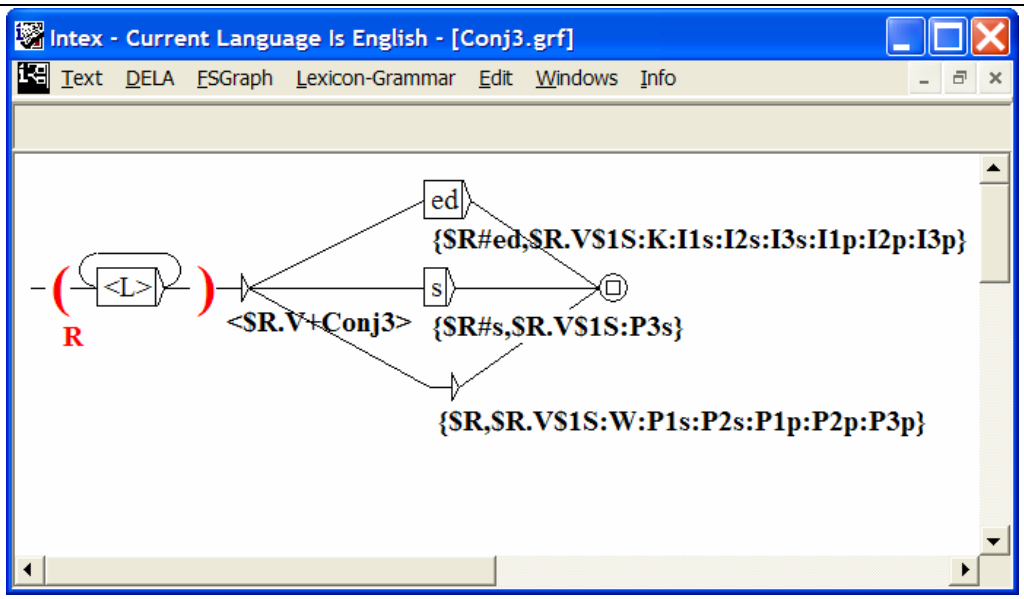


Figure 77. A morphological graph for the inflection

This graph uses the lexical constraint <$R.V+Conj3> together with the DELAS-type dictionary in order to insure that only verbs that are listed in the dictionary, and conjugate the right way, will be recognized. For instance, the word form “helps” is recognized by this graph (through the path in the center); variable $R then stores the prefix “help”; INTEX then checks that “help” is an lexical entry associated with the information “V+Conj3” in the dictionary, then finally produces the resulting tag: “{helps,help.V+Conj3:P3s}”.

Other, more irregular or complex conjugation schemes would be handled by shortening the root -- which could even be the empty string in the case of highly irregular verbs, such as “to be”.

This functionality can be used to formalize languages that have a heavy morphology, without having to construct artificially huge DELAF-type dictionaries (or maybe, even without any DELAF dictionary at all). It can be used also for Romance languages, to reduce the size of DELAF dictionaries.

VI. Syntactic analysis

Tokenization, Morphological analysis and the identification of simple and compound words, and frozen expressions generally yield ambiguous results. In chapter 13, we will see how to eliminate certain levels of ambiguity by using local grammars. In Chapter 14, we present the syntactic parser of INTEX, that can produce a tree representation of the syntagmatic structure of sentences, that is independant from the structure of the grammar.

Chapter 13. ELIMINATING LEXICAL AMBIGUITY

Within INTEX, the idea of *ambiguity* is a precise technical notion:

A form (simple or compound) is ambiguous if and only if it is associated with more than one different lexical element, in one or several of the lexical resources.

In general, after having applied the dictionaries and the lexical graphs to a text (**Text > Apply lexical resources**), numerous simple and compound forms can be associated with more than one different linguistic element. Here are some examples of ambiguity:
-- the form *RELEVE* corresponds to two lexical entries of the DELAF dictionary of French; *relevé* and *relève*;

-- the lexical entry *place* is associated with two different lexical elements in the DELAF dictionary: the feminine singular noun *une place* and the conjugated form of the verb *placer*;

-- the sequence *carte bleue* corresponds either to the simple word *carte* followed by the simple word *bleue* ("*une carte de couleur bleue*"), or to the compound word entry of the DELACF *carte bleue* ("*une carte bancaire*");

-- the form *des* corresponds either to the plural (ex. *Luc mange des fruits*), or to the contraction of the preposition *de* and the article *les* : *la chambre des enfants* (= *la chambre de les enfants*).

In general, it is not possible to automatically eliminate all lexical ambiguity limiting oneself to a phrase, by phrase analysis. For example, outside of certain contexts, the following phrases are incomprehensible:

Il y a une carte bleue dans le tiroir (*a credit card*? or *a blue map, or a blue business card, or a blue postcard*)

Luc a volé Marie (meaning *Luc stole 100F from Marie*, or *Luc stole Marie from Paul*)

Fortunately, numerous occurrences of lexical ambiguity can be eliminated automatically by using relatively simple mechanisms. INTEX provides several such mechanisms for disambiguation, from the dialogue box **Text > Disambiguation** :

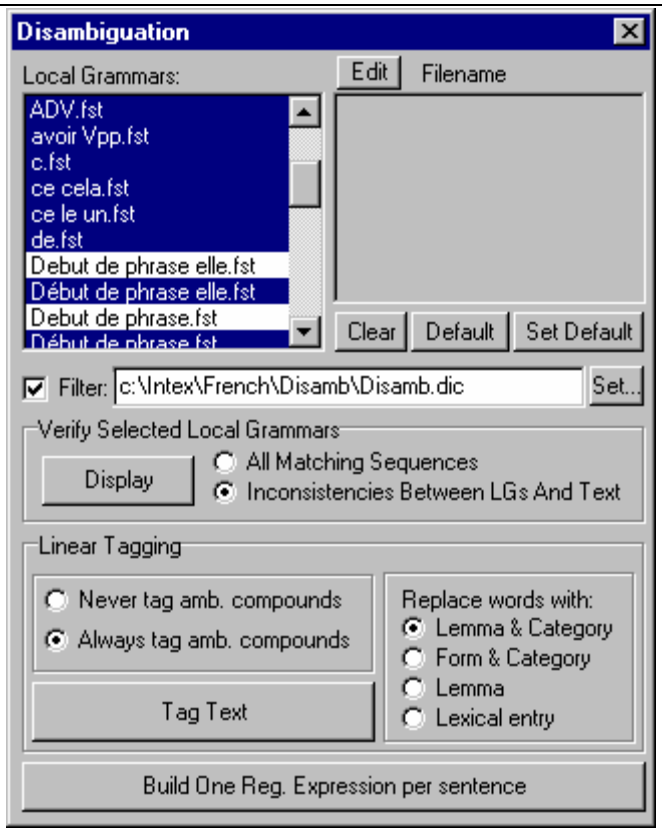


Figure 78. Lexical disambiguation

13.1. Disambiguation using a "filter" dictionary

The mechanism used to eliminate non-autonomous forms beginning with the preprocessing stage (**Text > Preprocessing**) can be generalized to all non-ambiguous compound words. For example, the following compound words occur frequently in texts:

```
c’est-à-dire, peut-être, quelqu’un, quelque chose
```

They do not have non-autonomous constituents, therefore, INTEX will systematically suggest two analyses: compound word (eg. The adverb *peut-être*), or the sequence of simple words (eg. The verbal form *peut* followed by the noun or verb *être*).

This being said, these words are not ambiguous. Since we don’t process them during the pre-processing stage, INTEX will break them up into simple forms (for example, "c", "est", "à", "dire"), and in so doing, systematically introduces artificial ambiguity. To avoid this problem, we can store these words in the "filter" dictionary from the Disambiguation dialogue box (the default *Filter Dictionary* is **Disamb.dic**).

Note that the mechanism handling degrees of prioritization within lexical resources can also be used in this module: for example, simple words that are generally ambiguous because they possess certain meanings, but which are never ambiguous in a given context, can, advantageously be stored in the "filter" dictionary.

Note: You are able to maintain more than one filter dictionary, depending on the domain or the application. All you need to do is store them in the **Disamb** folder of the current language.

13.2. Disambiguation using local grammars

Within INTEX, we can construct local grammars for the purpose of eliminating the ambiguity of certain words by specifying some of the contexts in which they appear. Transducers, which will recognize these particular contexts, represent local grammars and associate the lexical constraints, used to destroy lexical hypotheses, with the identified sequences.

A simple example of a local grammar of disambiguation is the following: Consider the sequence "*C’est*", very frequent in texts. This sequence is ambiguous for based on 9 possibilities. After having consulted the lexical resources of INTEX, we find the following lexical entries:

```
C,C.DET+CR=100
c,ce.PRO:ms
c,c.N:ms
est,est.A:ms:fs:mp:fp
est,est.N:ms
est,être.V:P3s
```

"*c"* is a roman numeral (100), a noun (representing the third letter of the alphabet) or the pronoun *ce* elided; "*est"* is a noun (point on the compass), an invariable adjective or a conjugated form of the verb *être*.

The following elementary transducer can be used to disambiguate this sequence:

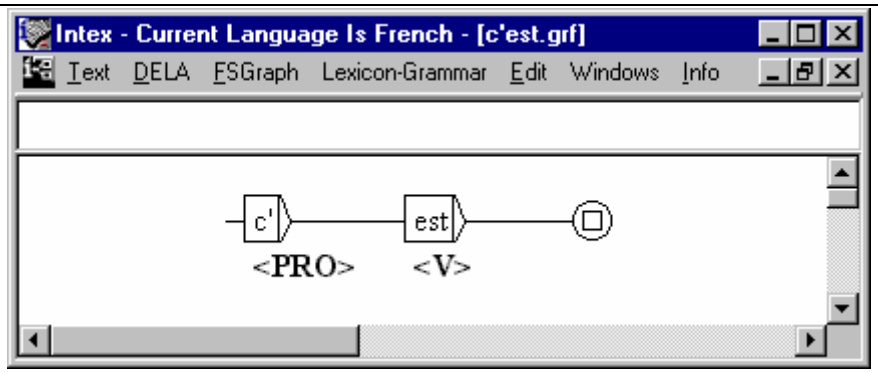


Figure 79. A local grammar of disambiguation

each time that such a sequence is encountered, the transducer produces the lexical constraints <PRO> for "*c"* and <V> for "*est*". Only the compatible lexical entries with the specified lexical constraints are taken into consideration:

```
c,ce.PRO:ms
est,être.V:P3s
```

The sequence is therefore completely disambiguated. A similar example can be seen in the following graph:

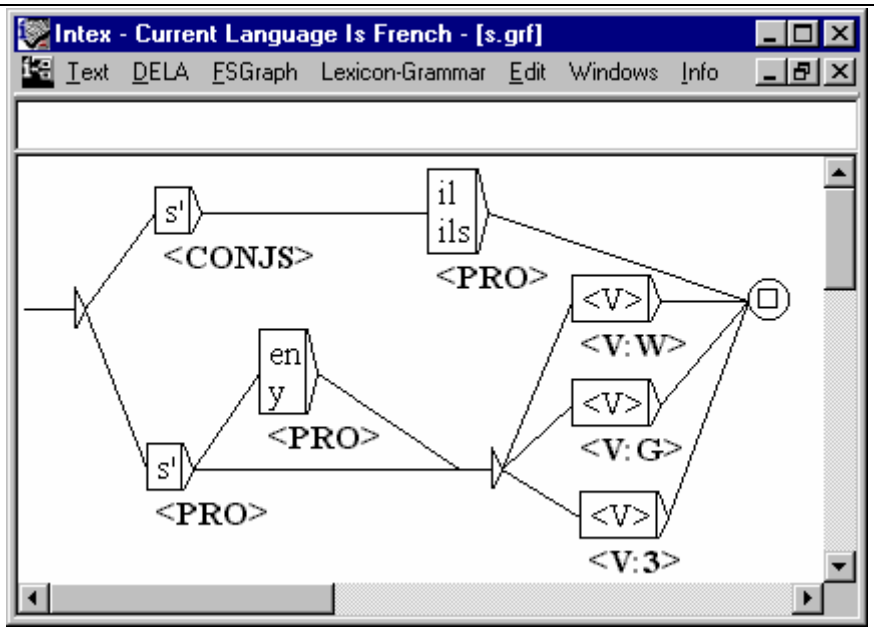


*Figure 80. Disambiguation of "*s'"

When we apply this transducer, the occurrences of the form "*s'"* followed by a pronoun "*il"* or "*ils"* are treated as the conjunction of subordination "*si"* ; When they

are followed by a verb (potentially preceded by the forms "*en*" or "*y*"), they are treated as the pronoun "*se*". Note that in sequence such as:

s'en donne

the three forms "*s'*" (*si* or *se*), "*en*" (the preposition or the preverbal pronoun) and "*donne*" (the feminine noun or the verb conjugated in the first or third person present of the indicative, or to the second person of the imperative) are, all three, completely disambiguated.

The constraints (produced by the transducer) must be synchronized with the words from the text (entries of the transducer). If we do not want to apply a constraint to a given form, we must associate the empty constraint "<E>" with it. For example:

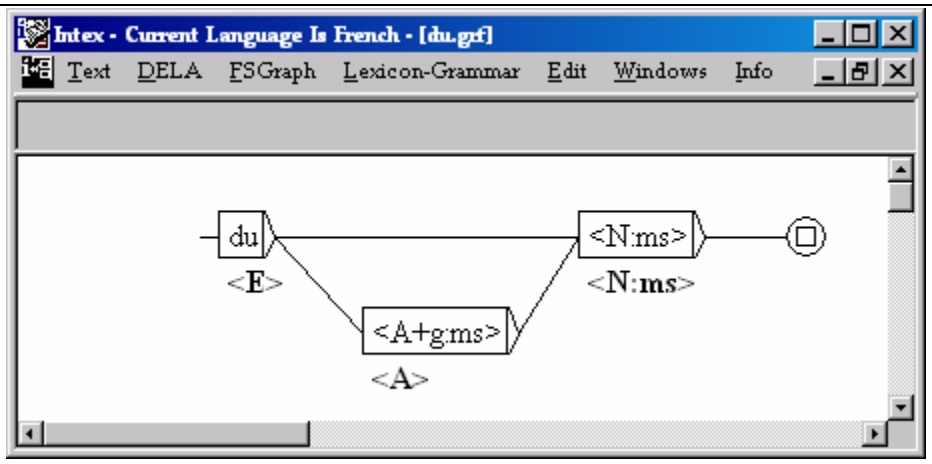


Figure 81. disambiguation to the right of "du"

Generally, graphs of disambiguation are more detailed because the minimal contexts, which must be well described so as to disambiguate without introducing error, are more complex. For example, the following graph is part of a collection of grammars that describe the sequences of preverbal particles, very constrained in French. These grammars allow us to resolve numerous cases of ambiguity because the majority of these preverbal particles occur frequently and are often ambiguous. (for example *le*, *la*, *les*, *leur* can be determiners or pronouns).

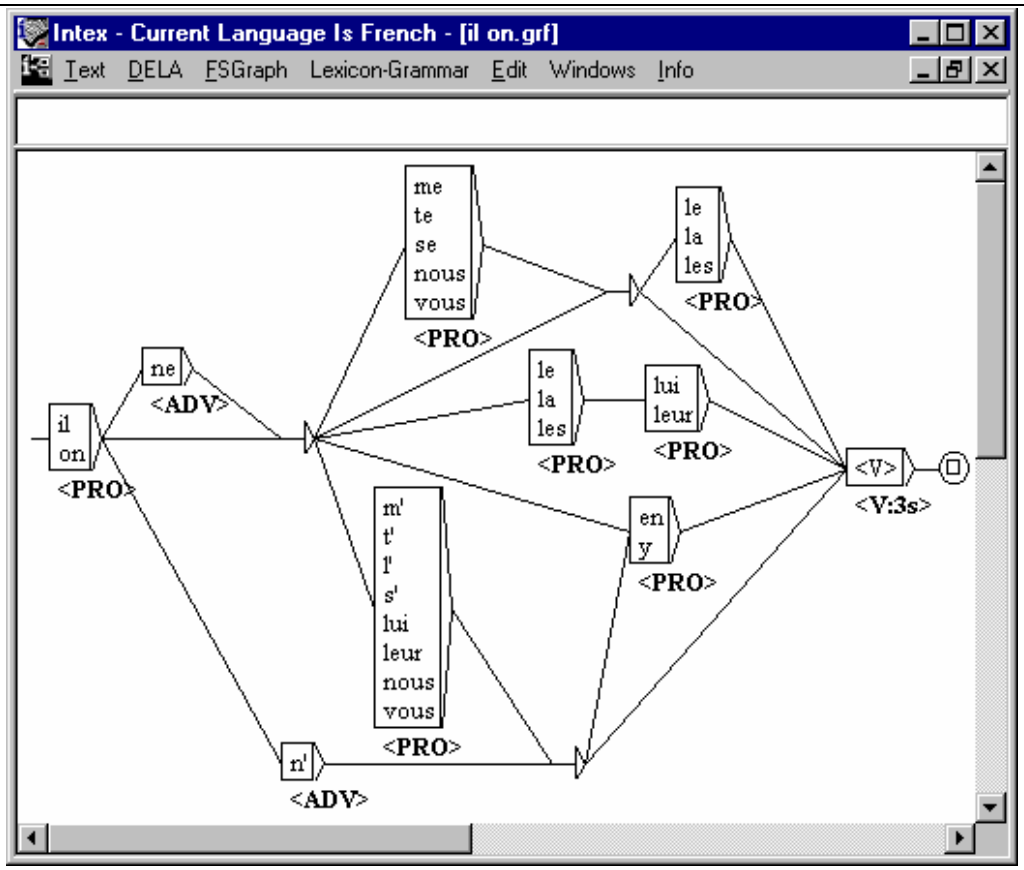


Figure 82. Preverbal pronouns

It is possible to construct "perfect" disambiguation grammars, meaning grammars that introduce no error into the texts.

For certain applications, we could add "imperfect" disambiguation grammars, either because they will only be applied to specific texts (for example, technical texts as opposed to general language texts), or because they allow disambiguation in a great number of cases and only introduce error so infrequently that it is considered negligible.

13.3. Tagging a text

Tagging a text consists in replacing, in the text, all the forms that are not ambiguous by the corresponding lexical entry, written between brackets *(French accolades specifically)*. The non-ambiguous forms of the text are either: forms corresponding to a single lexical entry from the dictionaries applied to the text, forms found in the filter dictionary or sequences that had been disambiguated by the application of one or more local grammars.

Generally, it is not possible to eliminate every lexical ambiguity from a text (cf. the discussion from the beginning of the chapter), so texts that are processed using INTEX are partially tagged. The texts are therefore sequences of simple forms and tags, which we are able to see, by selecting the "**Display tags**" function (at the top and to the left of each text window).

Within INTEX, the operation consisting of applying the filter dictionary and local grammars to alleviate cases of lexical ambiguity, then tagging the text, is called: "**Linear tagging"**. This name reflects the idea that once this type of tagging has been done, the text is comprised of a linear flow of lexemes that are either forms or lexical entries rather than the other two forms of text representation in the text: the rational expression and the text transducer.

When we "**Tag text"**, two series of options are available to us:

(1) **Never tag ambiguous compounds vs. Always tag ambiguous compounds**

Given that it is not possible to disambiguate ambiguous compound words (remembering the example "il y a un cordon bleu dans la cuisine"), two solutions are possible:

-- either the ambiguous compound words are recognized and tagged as such; e.g. all occurrences of the sequence *cordon bleu* will be treated as the human noun signifying *bon cuisinier (good cook)*;

-- or the ambiguous compound words are ignored; e.g. all occurrences of the sequence *cordon bleu* will be treated as the concrete noun signifying *un cordon de couleur bleue (a cord which is blue in color).*

Of course, these solutions are both incorrect in theory (by definition of the ambiguity of compound words!). They correspond to two purely pragmatic choices:

-- the first solution often gives satisfactory results, especially when recognized compound words are technical terms, for example *système d'exploitation*, *micro-ordinateur*, *traitement de texte*, etc. Unless one in making fairly complex plays on words (that we tend not to do in the first place), it is reasonable to treat all recognized terms as being real terms;

-- the second solution is that adopted by the scientific community at large, interested text tagging and which, in great majority, ignores the problems posed by the great frequency of compound words in the texts *(I won't go further than that).*

(2) **Replace words with...**

The second series of options concerns the format of tags inserted in the text. The non-ambiguous forms can be replaced by their lemma (which allows the lemmatization of the text), or their lemma and morpho-syntactic category (the result of which is often

used by researchers who perform a statistical analysis of the texts), or the entire corresponding lexical entry.

The lemma associated with each lexical entry can have various functions; by using the option "**Replace words with lemmas**", we can translate terms in the text if the dictionaries associate the terms with their translation:

e.g. `système d'exploitation,operating system.N`

or phoneticize the text, if the lexical entries are of the type:

`pharmacie,farmasi.N`

13.4. Rational expression within the text

The primary problem with linear tagging is that all of the forms that remain ambiguous are left as is, and we don't see the ambiguities in the text. The solution is to represent each sentence of the text by a regular expression, in which, as before, the non-ambiguous forms are replaced by the corresponding lexical entry, and the ambiguous forms are replaced by the separation of all of the corresponding lexical entries. For example, if the following text:

`Il donne la pomme`

Had not been disambiguated (i.e. correct upon consultation of the dictionaries), the rational expression of the text would be as follows:

```
{il,il.PRO:3ms}
({donne,.N:fs} + {donne,donner.V:P1s:P3s:S1s:S3s})
({la,le.DET:fs} + {la,le.PRO:fs})
({pomme,.N:fs} + {pomme,pommer.V:P1s:P3s:S1s:S3s})
```

The advantage of this representation, with respect to linear tagging, is that it allows us to represent the lexical constraints produced by the grammars of disambiguation, even when these have not produced a completely disambiguated result. For example, a local grammar can produce the constraint wherein the determiner *la* must be followed by the noun *pomme*, whereas the preverbal pronoun *la* (and *il*) must be followed by the verb *pommer* (*donner*). After the application of this local grammar, the rational expression of the text would be as follows:

```
{il,il.PRO:3ms}
{donne,donner.V:P1s:P3s:S1s:S3s}
({la,le.DET:fs}
{pomme,.N:fs}
+
{la,le.PRO:fs}
{pomme,pommer.V:P1s:P3s:S1s:S3s})
```

The second advantage of this representation is that it allows representation of the ambiguous compound words; for example, the text "*il donne la pomme de terre*" would be represented by the following expression:

```
{il,il.PRO:3ms}
{donne,donner.V:P1s:P3s:S1s:S3s}
(
{la,le.DET:fs}
({pomme de terre,.N:fs} + {pomme,.N:fs} {de,.PREP}
{terre,.N:fs})
+
{la,le.PRO:fs}
{pomme,pommer.V:P1s:P3s:S1s:S3s} {de,.PREP} {terre,.N:fs}
)
```

The inconvenience of this representation is that when compound words follow one another (as in *femme de chambre d'hôtel* for example), this notation quickly become illegible.

Chapter 14. SYNTACTIC ANALYSIS

14.1. Constructing the text transducer (FST)

Figure 83. Constructing the text FST

The linearly tagged text cannot furnish a correct representation of the lexical analysis since ambiguity between compound words and sequences of simple words are not included therein, as is the case for the non linear lexical constraints such as: {la,.DET} {donne,.N} + {la,.PRO} {donne,.V} (if *la* is a determiner, then *donne* is a noun; if *la* is a pronoun, then *donne* is a verb).

The representation of each sentence of a text by a rational expression allows us to be aware of these phenomena, but is not ideal because it requires a lot of copies and is practically illegible when it comes to representing series of ambiguous compound words that follow one another.

Representation of text by FST

For these reasons, INTEX represents text issuing out of lexical analysis by a finite transducer in which:

-- each linguistic unit is represented by a transition tag,
-- each route between the initial and terminal nodes corresponds to a possible reading of each sentence.

The text transducer is constructed by initiating the command **Text > Construct FST-Text**. For example, the text:

```
il donne la pomme de terre
```

is represented by the following transducer:

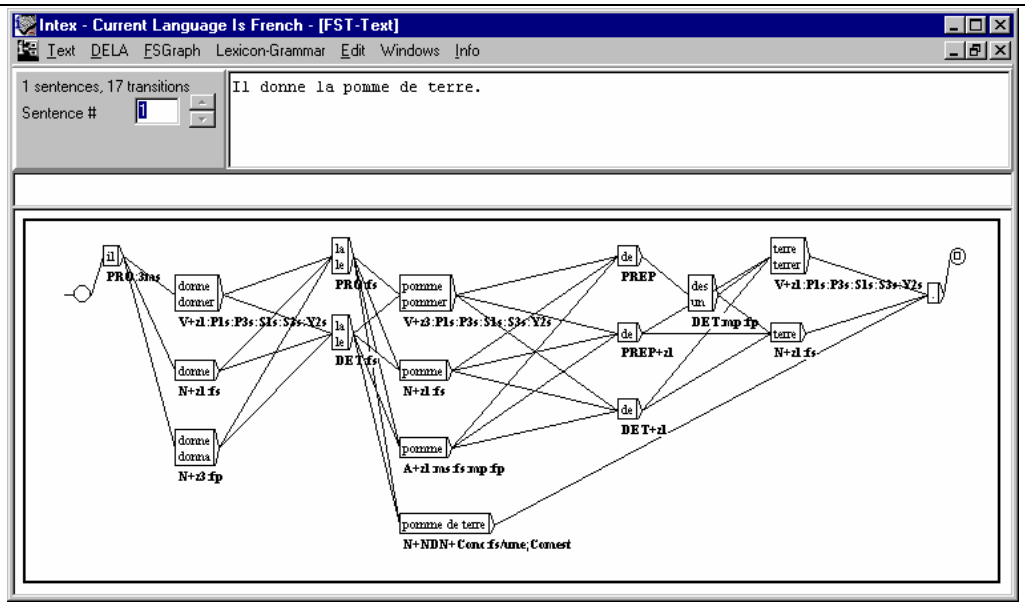


Figure 84. Representing a text by transducer

il is a pronoun; *donne* is either a form of the verb donner, a feminine noun or the plural of the noun *donna* ; *la* is a pronoun or a determiner; *pomme* is either a form of the verb *pommer*, a noun, or an adjective; *de* is either a preposition, a determiner, or the contraction of the preposition *de* and the plural determiner *des* ; *terre* is either a form of the verb (*se*) *terrer*, or a feminine noun; *pomme de terre* is either a compound noun or a sequence of three simple forms. This transducer contains 60 possible paths between the initial and terminal nodes; each path corresponds to a lexical analysis of the sentence.

Representing frozen expressions

Frozen expressions, as with compound words, are ambiguous from the outset; they are therefore represented in parallel paths within the text FST. For example, is we apply the transducer **perdre la raison** to the text:

```
Ne perd jamais la raison.
```

The text FST becomes:

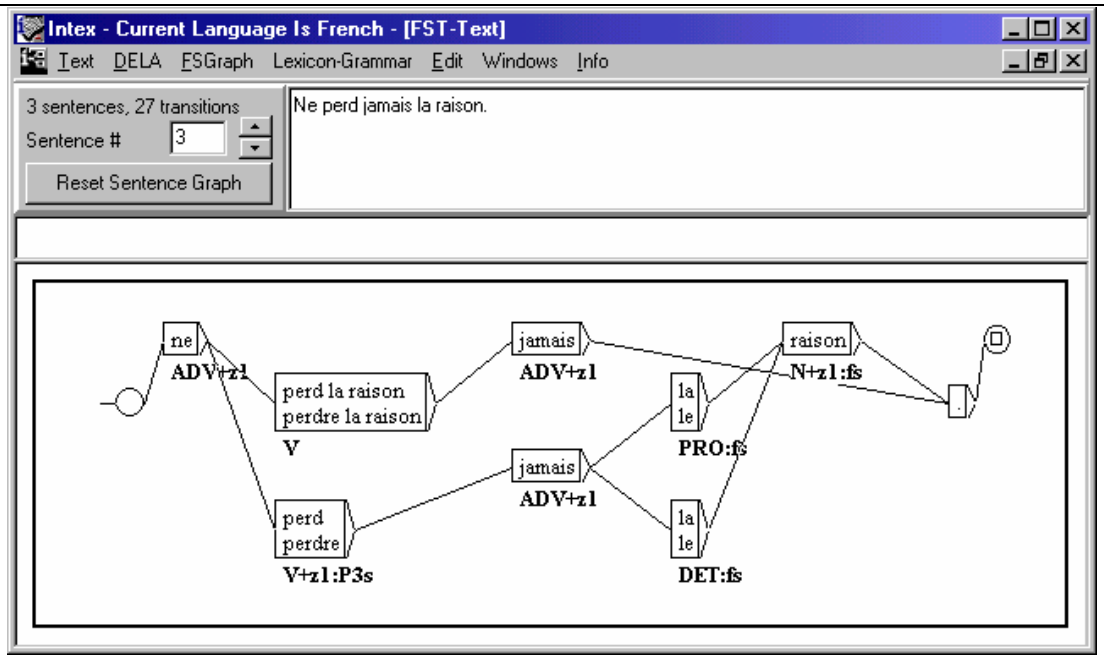


Figure 85. Representation of a frozen expression in the text FST

Note the displacement of the adverb *jamais* in the upper path.

In the text FST, the frozen expressions are represented as atomic linguistic units (within a single node), exactly as simple or compound words. This requires separating the free constituents from their expression, just as we extracted the subject of the expression "*N perd la raison*". Consider the following expression:

C0 *La moutarde monte au nez de N*

If we apply this expression's FST, or even that of the entire **C0** table, to the text "*La moutarde monte au nez de Luc*", we will obtain an FST similar to the following:

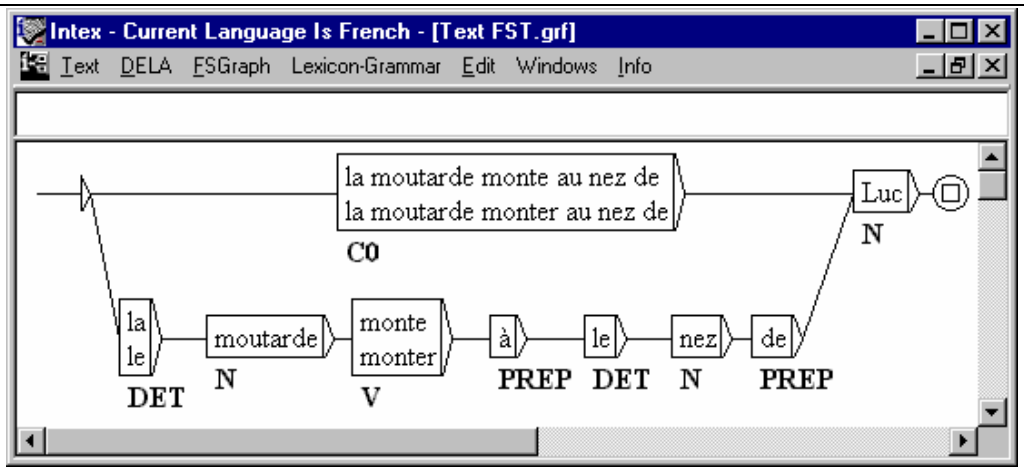


Figure 86. Representation of a frozen expression

(the ambiguities have not all been indicated). The lower sequence in this transducer corresponds to the literal interpretation (i.e. where the frozen expression has not been taken into account); this sequence is grammatically "standard"; a syntactic analyzer would have no problem constructing the following tree:

$$(\text{DET N})_{\text{N0}}\ \text{V}\ (\text{PREP DET N}\ (\text{PREP N})_{\text{Modif}})_{\text{N1}}$$

On the other hand, the sequence represented by the upper path is "non-standard", since it describes a sentence containing a linguistic unit of the type **C0** followed by a noun. If a syntactic analyzer is to validate this sentence, we would have to add the sequence "**C0 N"** to the grammar, beside dozens of other artificial constructions of this sort, since each table of frozen expressions corresponds to a specific basic structure, and a half-dozen or so other transformations (passive, extraction, pronominalization, etc.).

This solution is not viable if we want to independently develop a syntactic analyzer and a description of frozen expressions. What's more, the great majority of frozen expressions have a syntactically "standard" internal structure; it would be a shame to not take this into account in the syntactic module.

The solution that we suggest is to not modify the sentence structure, which necessitates attaching the lexical information to one of the constituents of the frozen expression (for example the main verb), more so than to the entire expression. The preceding text FST would then simply become:

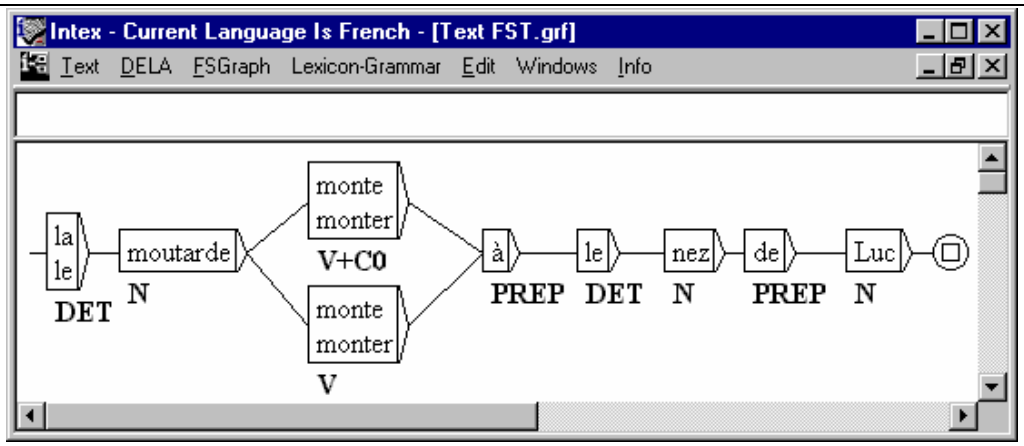


Figure 87. Alternative representation

It then becomes the responsibility of the syntactic analyzer to verify whether or not the context of the verb *monter* "**C0**" in the text FST is compatible with the information presented in the **C0** table. This solution comes down to treating frozen expressions as free verbs in the syntactic module.

Standardization of the text FST

Certain ambiguities could not be represented in the tagged text (in linear format) because they involve sequences of one or more simple forms. For example, such is the case of the ambiguities with the form *de*, which could represent a preposition (eg. *Luc vient de Paris*) or a contraction *de = de des*, (eg. *Luc rêve de vacances*, contraction of: *Luc rêve de des vacances*).

These ambiguities involve a half-dozen or so forms in French. They are naturally represented in parallel paths in the text FST. To add these paths, we use the standardization FST (the file **Norm.fst** must be present in the folder of the active language). Here is the default transducer used by INTEX for French:

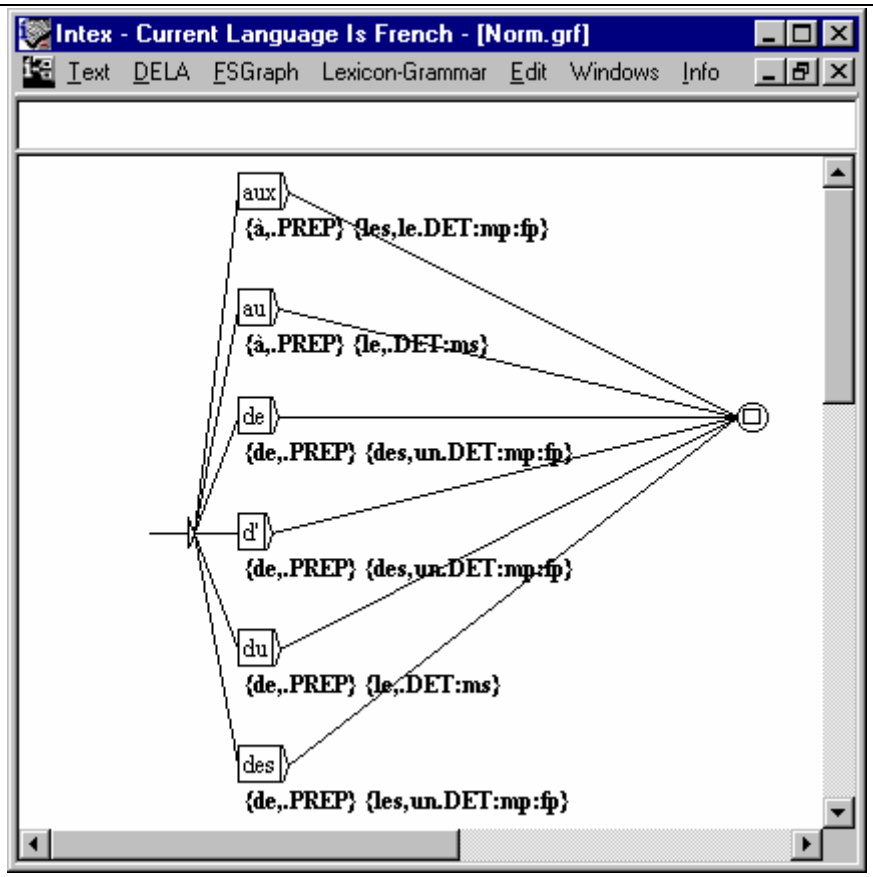


Figure 88. Standardization graph for the text FST

This transducer resembles the lexical transducers of simple words, compound words or frozen expressions, since, like them, it is used to add parallel paths to the text FST. But, contrary to lexical transducers that only add linguistic units (nodes) to the text FST, this transducer gives the possibility of adding paths comprised of more than one linguistic unit.

Caution: if we apply the standardization FST, you must verify the coherency (and the non-redundancy) of information produced by this transducer, against that information produced by the lexical resources. In particular, you must remove (using the filter dictionary for example) the lexical entries "**PREPDET**" found in the French DELAF.

Using the vocabulary of the text

The vocabulary of the text is represented in three dictionary files: **DLF** (simple words from the text), **DLC** (compound words from the text) and **DLE** (frozen expressions from the text). When constructing the text FST we can select the files that should be taken into consideration.

We can also specify the name of the "filter dictionary" (in general, the same as that which is used by the text tagging module), which contains disambiguated simple or compound words.

The text FST exactly represents the results of the lexical analysis; once constructed, we can perform a syntactic analysis. From this point on, the basic units that will be processed are lexical entries.

14.2. Where text FST and local grammars meet

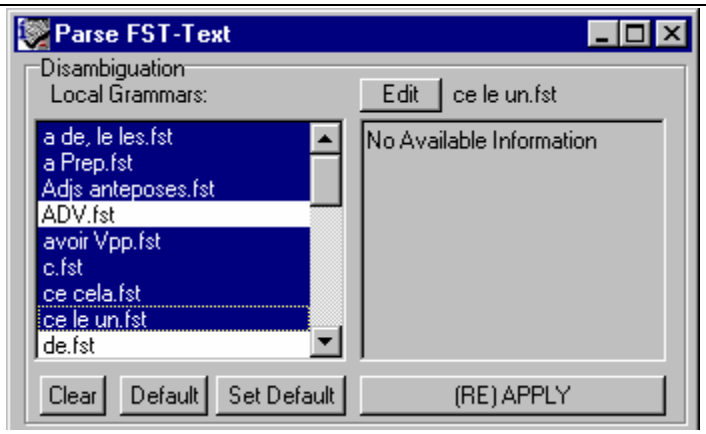


Figure 89. Syntactic analysis

We apply disambiguation grammars to the text, which are generally the same as those used by the tagging module (in the **Disamb** menu).

Differences between the elimination of ambiguity during the tagging and syntactic analysis processes

One important difference between these two programs is a result of the fact that in the text FST, the units of treatments are linguistic units (represented by non-ambiguous lexical entries), while the tagging module addresses ambiguous forms.

Here is an example of a grammar that wouldn’t provide the same results in the tagging module as in the syntactic module:

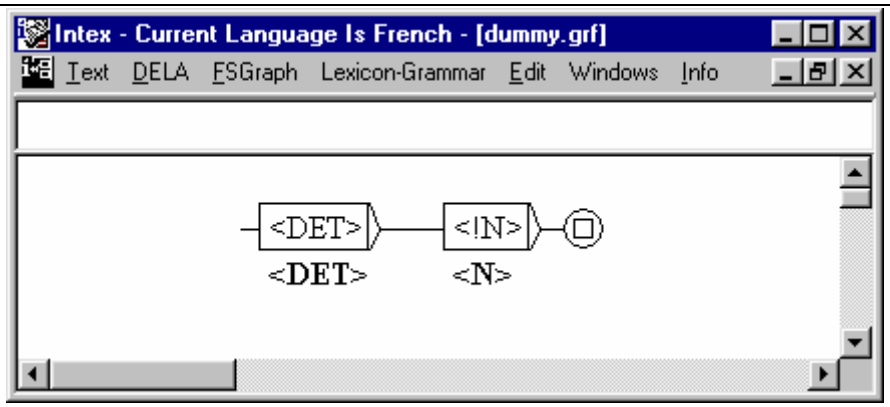


Figure 90. Example of a grammar

In the tagging and **Disambiguation** modules, this transducer must be interpreted in the following manner:

If we find a determiner followed by a form that could be something else besides a noun, we impose the fact that this form be a noun

This transducer (which often provides false results) works in the following manner.

-- if a non-ambiguous form follows a determiner, the transducer will not recognize the sequence (since the non-ambiguous noun is not recognized by the symbol <!N>) and therefore does nothing;

-- if a determiner is followed by a form that is not a noun, the FST will recognize the sequence but will be unable to apply the <N> constraint on that form; Consequently, the FST will not do anything;

-- if a determiner is followed by a form that is associated with a nominal lexical entry as well as to one or more non-nominal lexical entries, the FST will recognize the sequence (since the form is recognized by the symbol <!N>), and oblige this form to be interpreted as a noun (the <N> constraint is therefore applied).

In summary, each time that a determiner is followed by an ambiguous form that could be a noun, this transducer excludes any other possibility and will treat it ONLY as a noun.

In the syntactic analysis module (**Parse**), the units of processing are not forms (which can be ambiguous), but non-ambiguous linguistic units, represented by lexical entries in the nodes of the text FST. A linguistic unit cannot simultaneously be identified using the two symbols <!N> and <N>. Consequently, this transducer has no use (it would make no sense).

The second difference between the two disambiguation modules lies in the fact that the text FST can portray more than one disambiguation solution for a given sequence, which is not possible in a tagged text (which only portrays a single possible sequence). For example, let's consider the following sequence: "*la donne*". *La* can be

a determiner, a masculine name (the musical note) or a preverbal pronoun; *donne* can be a feminine noun or a conjugated form of the verb *donner*. This sequence is represented by the text FST as follows:

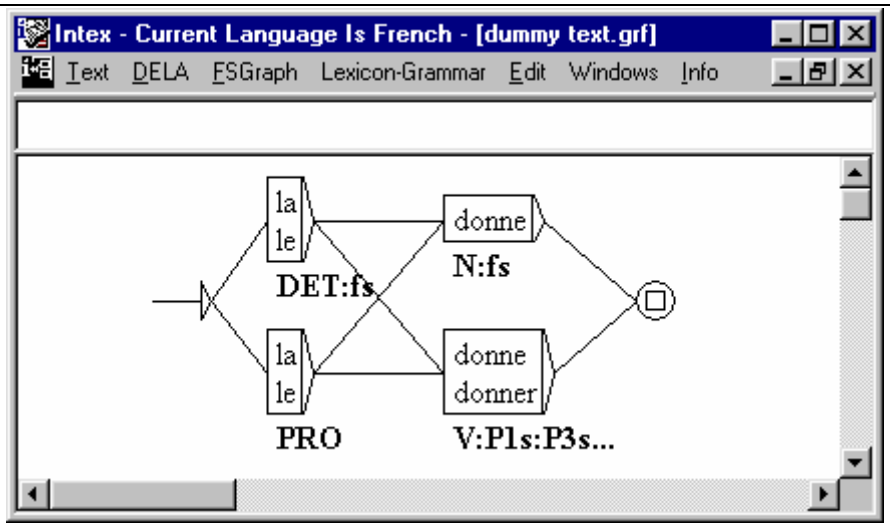


Figure 91. Example of ambiguous text

The following disambiguation grammar produces lexical constraints such as: if *la* is a determiner, then *donne* is a noun; if *la* is a pronoun, then *donne* is a verb:

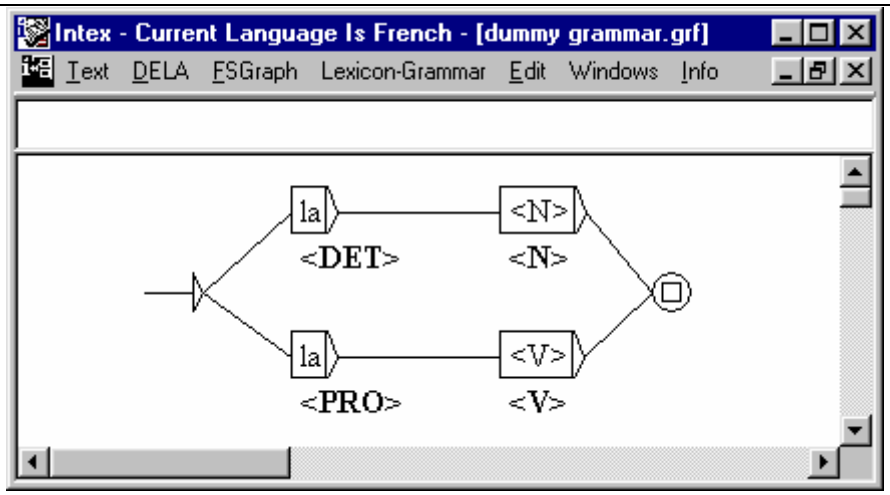


Figure 92. Example of a disambiguation grammar

If we use this disambiguation grammar in the tagging module, the resulting text would not be modified since the two forms *la* and *donne* remain ambiguous. On the other hand, in the syntactic module, the text FST would be:

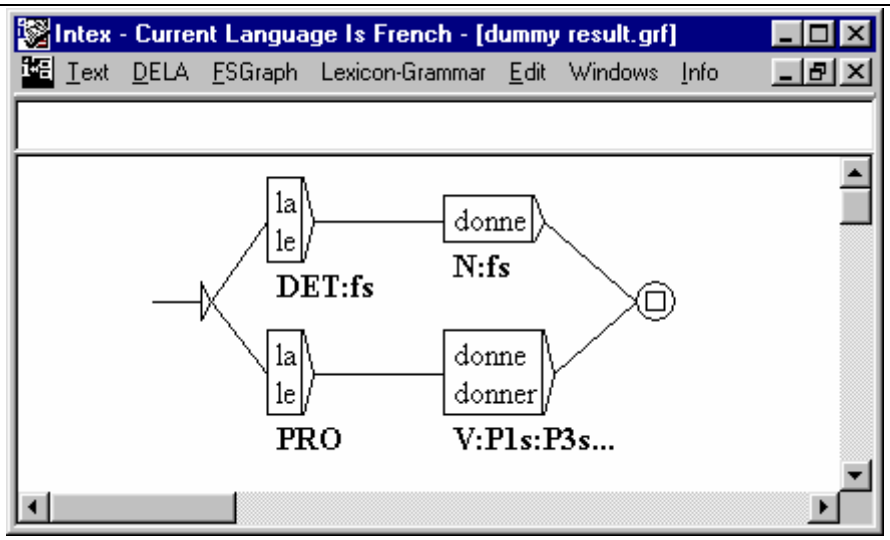


Figure 93. Text FST after disambiguation

14.3. Syntactic analysis of the syntagm

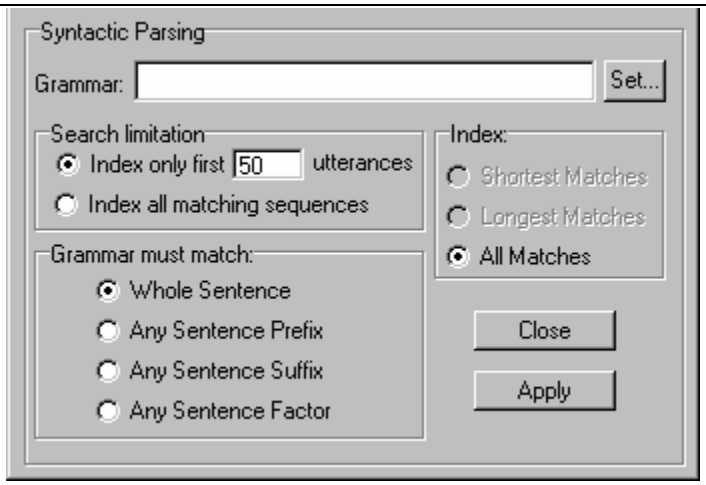


Figure 94. Syntagmatic analysis

The syntactic analyzer is similar to the capacity to search for morpho-syntactic motifs upon which we call from the dialogue box "**Locate Panel**". In both cases, we are describing a collection of sequences with the help of a series of graphs that we apply to the text.

The grammars applied in the syntactic module can represent a **Whole Sentence**, a **Sentence Prefix**, a **Sentence Suffix** or parts of the sentence placed anywhere (**Any Sentence Factor**).

Caution: if you select the **Whole Sentence** function, you must think to integrate end of sentence punctuation into the graph or grammar (period).

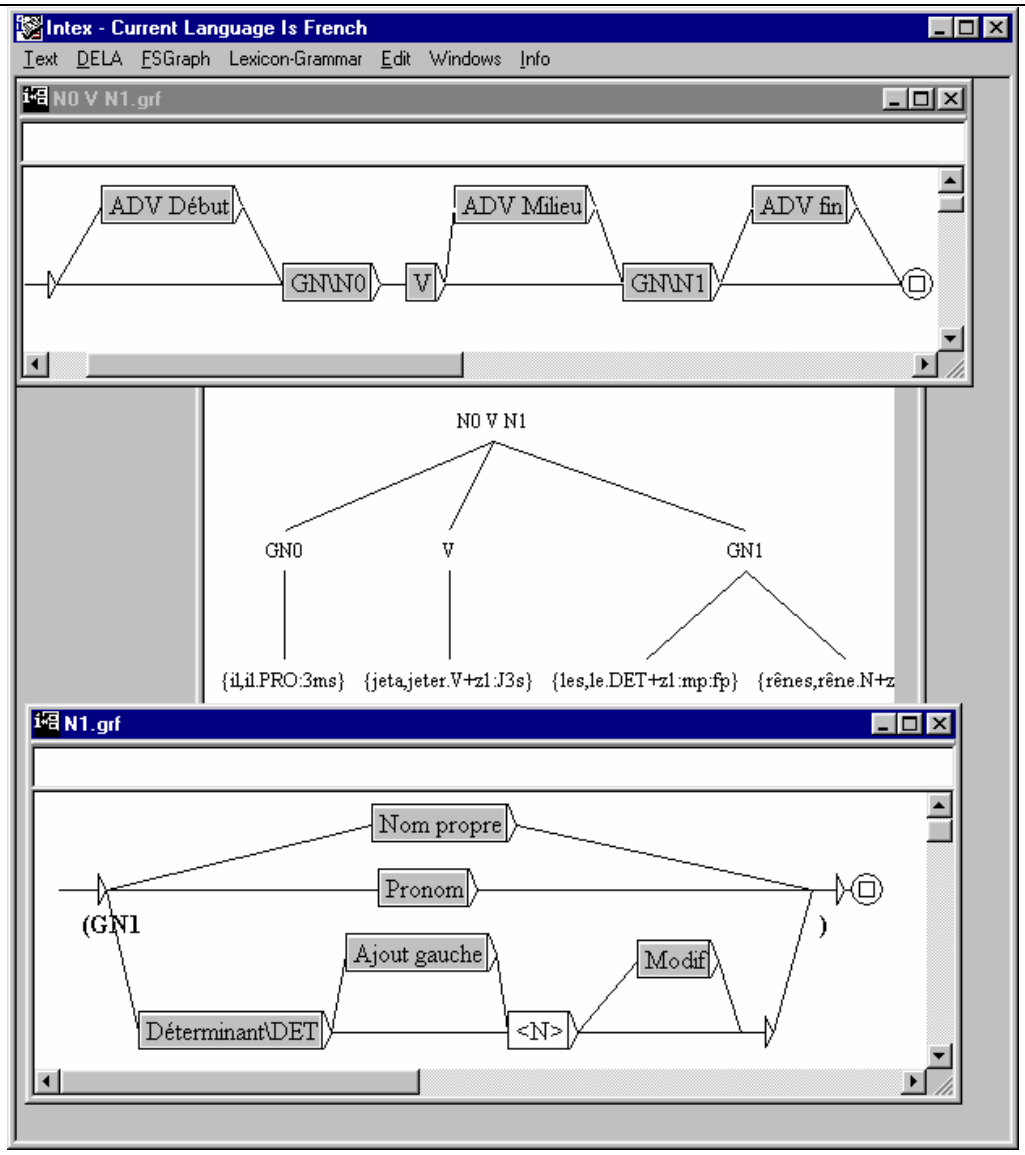


Figure 95. Syntactic structure of the sentence "il jeta les rênes"

The differences between the grammars used to search for motifs and those used to carry out syntactic analysis are as follows:

-- on one hand, the syntactic analyzer applies the grammar to the text FST, rather than linearly to the text itself. Consequently, the forms and lexical symbols of the grammar are always confronted with non-ambiguous linguistic units;

-- on the other hand, the results produced by the transducer of this grammar are used to construct a structured representation of the text, in general the syntagmatic structure of each sentence identified. The resulting structure is seen in the form of a tree.

The product of transducers, are tagged parentheses that represent the structure of recognizable sequences. For example, in the graph N1 of the preceding figure, the

parentheses tagged **GN1** are inserted around the sequences recognized by the five paths of the graph N1.

In relation to traditional syntactic analyzers, the syntactic analyzer of INTEX introduces two new characteristics:

(1) The text, which constitutes the entry of the syntactic analyzer of INTEX, is represented with a transducer; in which each linguistic unit is represented by a node, wherein the entry tag shows the form from the text and the exit tag contains the associated lexical information. This type of representation allows us to process the four types of linguistic units (morphemes, simple words, compound words and frozen expressions) in an identical fashion in terms of syntactic analysis, and to represent all types of lexical ambiguity (between simple words, between simple words and compound words, etc.) in a homogenous manner, by parallel paths.

(2) The grammar used by the syntactic analyzer is itself a transducer, which associates structural information to the identified sequences, in the form of tagged parentheses. For example, let us consider the following graph GN0:

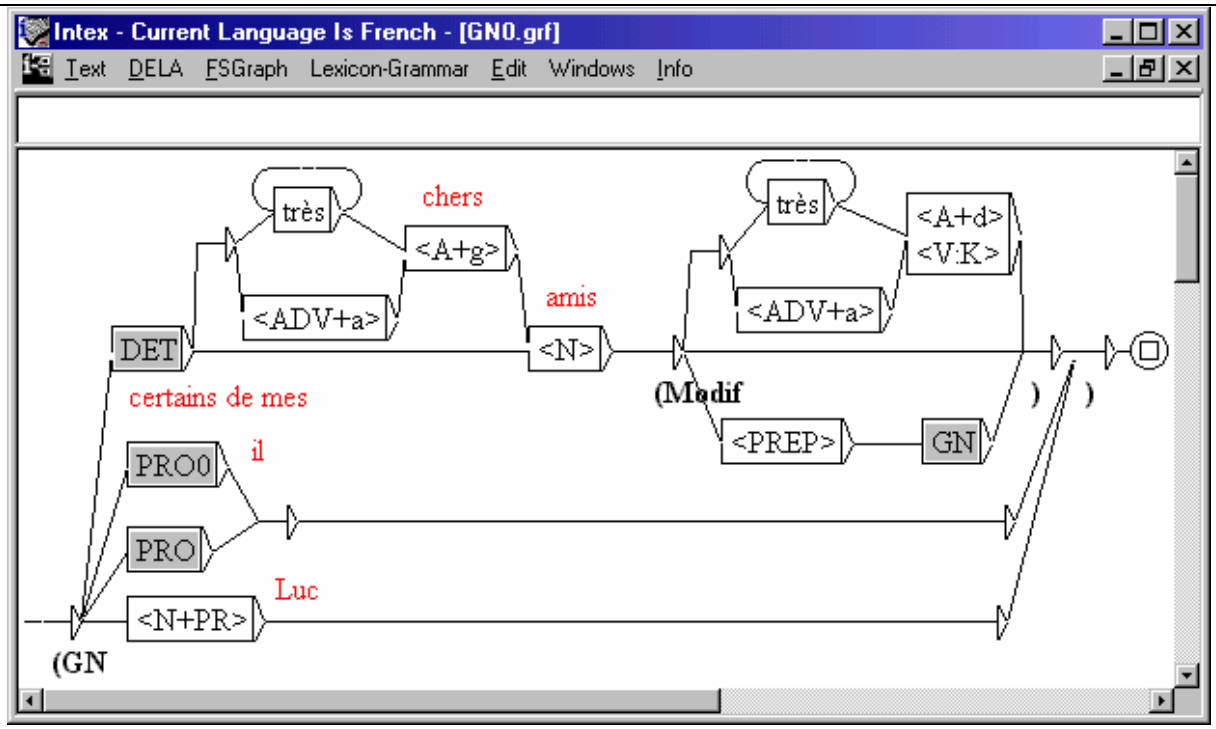


Figure 96. A description of non-sentential nominal groups

We have already mentioned the graph DET; the graph PRO0 describes the preverbal subject pronouns (e.g. *je, c', il*) and the simple pronouns (*cela, elle*, etc.) that can appear in the subject position; PRO represents structures that have the value of a pronoun: compound pronouns (*une personne, quelque chose*), Dadv (*beaucoup*), Dnom (*une foule*), Dadj (*certains*) and Dnum (*trente-trois*), and the structures with determiners (*certains d'entre eux, les trois premiers*), excluding preverbal pronouns.

The graph GN is similar to GN0, but does not recognize the personal pronouns, and describes the nominal groups that can appear anywhere in the sentence (where GN0 describes the subject nominal groups); the lexical symbols <A+g> and <A+d> designate adjectives that can be placed respectively to the left and to the right of a noun; the lexical symbol <ADV+a> (not yet coded in the DELAF dictionary) designates the adjectival adverbs (e.g. *peu*, *stupidement*, excluding verbal adverbs such as *dans l'intimité la plus stricte*).

Phrasal nominal groups (ex. *l'espoir que Luc vienne*) or phrasal groups containing relative propositions (ex. *l'homme que j'ai vu*), participle constructions (ex. *une femme connaissant bien son métier; un homme jugé responsable*) or constructions with an infinitive (ex. *le problème à résoudre*) are not treated here. See also Cf. M. Salkoff 1973 for a more complete description of nominal groups and sentences in French.

The transducers GN0 and GN (which insert the tagged parentheses **GN** and **Modif**) and DET (which inserts the parentheses **DET**), produce the structure of the nominal groups identified by this grammar; this structure is therefore independent of the structure of the grammar itself, in other words the hierarchy of embedded graphs in the description. For example, the structure of the nominal group:

```
La plupart de mes très chers amis de cette école
```

is represented by INTEX in the following manner:

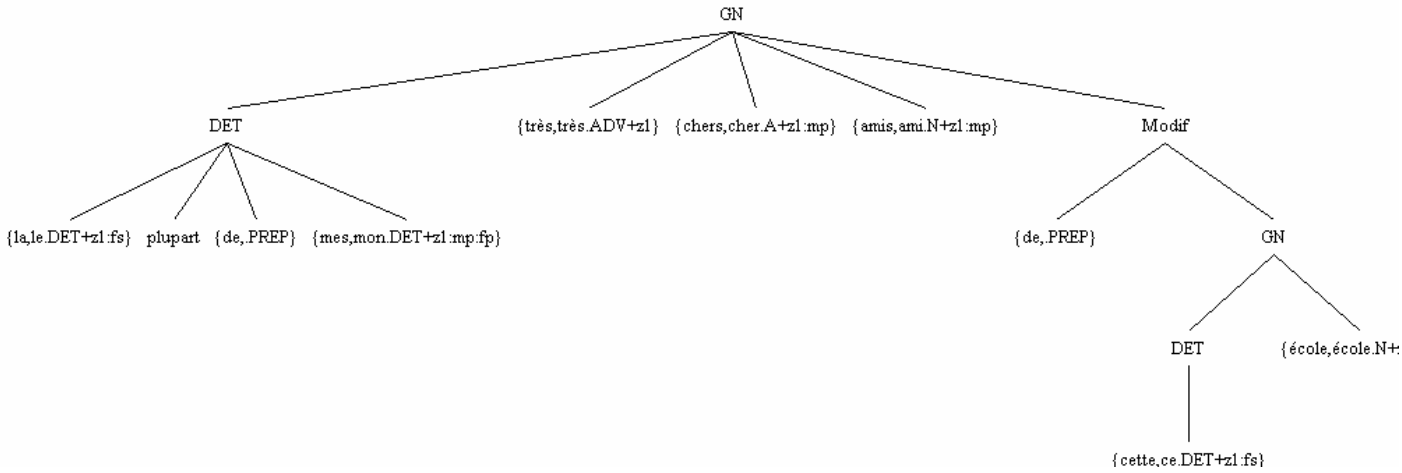


Figure 97. Syntactic structure of the nominal group

While the complex determiner "la plupart de mes" was recognized by being passed through 5 embedded graphs (DET, Dnom, Dde, Ddéf, Ddéf:p), this sequence is represented in a "flat" way in the sentence tree. Similarly, the graphs GN and GN0 are two variants of the nominal groups for which there is little purpose in representing them at two different levels; finally the graph GN is responsible for the production of the structural information **Modif** for which there is little use in representing it by another embedded graph.

By rendering independent the structures of a grammar and a sentence, INTEX gives linguists the possibility to re-use "pieces of a grammar" constructed by others (e.g. DET, GN, DATE, Ins, etc.) without having to bother with their internal structure. The "pieces" of the global linguistic description can therefore be organized along lines of clarity, of re-usability, without consequence on the analysis itself.

In the normal mode of operation, on the structure produced by the grammar trandsducers is taken into consideration. The "**Debug**" mode, on the other hand is used to show the derivational tree produced during the analysis (this is the mode available in traditional syntactic analyzers).

VII. Advanced lexicons

In this section, we describe the INTEX tools that can help build, process and manage DELA-type dictionaries (chap. 15) and lexicon-grammar tables (chap. 16).

Chapter 15. DELA DICTIONARIES

The dictionaries used by the lexical analyzer (**Apply Lexical Resources**) are of the DELAF (or DELACF) type, which means that they must contain the inflected forms of simple or compound words, as it's entries.

In general, linguists do not directly construct these dictionaries, but construct DELAS (DELAC) type dictionaries, in which each entry is a lemma associated with a morphological description which will be used by INTEX to automatically inflect it; the result of that automatic inflection would be the corresponding DELAF (DELACF) dictionary.

INTEX contains tools which verify the format and the sorting, the automatic inflection and the compression of both dictionaries and transducers.

15.1. Verifying the format of a DELA dictionary

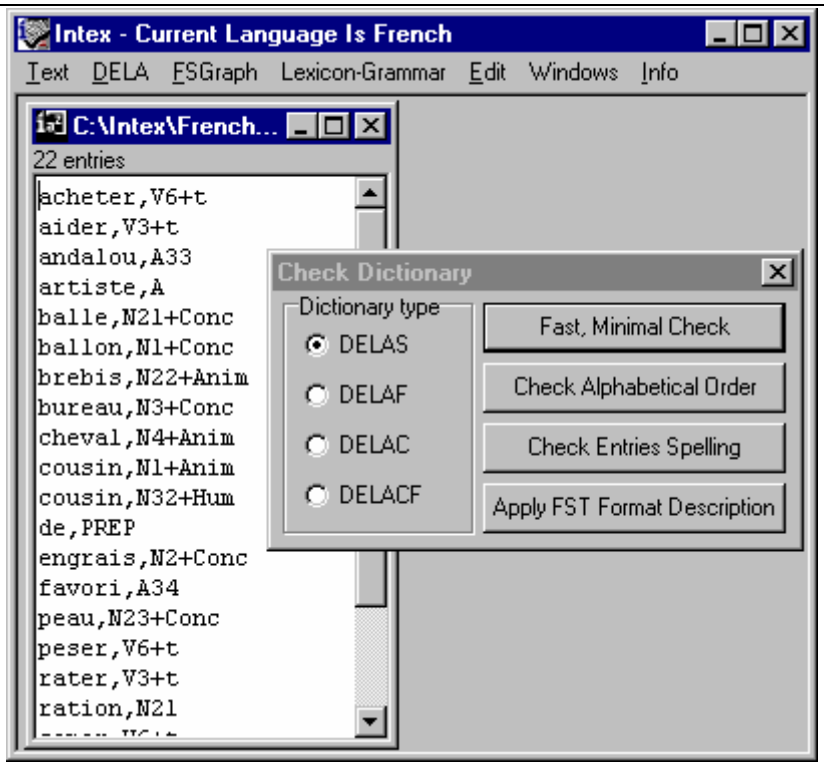


Figure 98. Verifying a DELAS

Four types of verification are available thanks to the function "**DELA > Check Format**" (Don't forget to load the dictionary first using the "**DELA > Open**" command).

Fast, Minimal Check

INTEX will verify that the open dictionary file does not contain any blank lines, then, depending on the type:

-- **DELAS**: each line must contain a simple form (i.e. a sequence of letters found in the alphabet of the active language), followed by a comma, a category name in capital letters, and eventually a number (indicating the inflectional class), one or more bits of syntactico-semantic information introduced by the character "+" or a comment introduced by the character "/";

-- **DELAF**: each line must contain a simple form (i.e. a sequence of letters found in the alphabet of the active language), followed by a comma, a simple form (could be empty), a period, a category name in capital letters, and potentially one or more bits of syntactico-semantic information introduced by the character "+", as well as one or more series of inflectional information introduced by the character ":" or a comment introduced by the character "/";

-- **DELAC**: each line must contain a text beginning with a letter, followed by a comma, a category name in capital letters; one or more bits of syntactico-semantic information introduced by the character "+" or a comment introduced by the character "/";

-- **DELACF**: each line must contain a text beginning with a letter, followed by a comma, a text (could be empty), a period, a category name in capital letters and potentially one or more bits of syntactico-semantic information introduced by the character "+", as well as one or more series of inflectional information introduced by the character ":" or a comment introduced by the character "/".

The value of this minimal verification (with respect to the full verification as per follows) is that it corresponds to the conventions adopted by all INTEX programs, regardless of the language or the intended application. In effect, while the choice of category codes, syntactic or semantic information, or inflectional codes, is free, there is still a minimal format which must be respected in order for INTEX to be able to read the dictionaries: we cannot use category codes in lower case letters (since there would be a conflict with the lemmas in lexical symbols), the characters "," and "." are forbidden in codes, etc.
The essential limits of the minimal verification are:

-- a DELAF dictionary which conforms to this verification should not contain inflectional codes (the DELAF is the product of the automatic inflection of a DELAS which is where the inflectional information is stored, they must therefore be removed);

-- a DELAC dictionary which conforms to this verification will not contain the necessary information for it’s own automatic inflection (but no INTEX program is currently capable of processing DELAC dictionaries).

Check Alphabetical Order

INTEX verifies that the alphabetical order of the dictionary entries is coherent with the alphabetical order described in the "**Alphabet**" file of the active language.

The command "**DELA > Sort Dictionary"** allows you to sort the dictionary.

Check Entries Spelling

INTEX applies to the dictionary to verify the dictionaries of simple words housed in the list "**def.lst"** stored in the folder "**dic-utils"** of the active language, as well as the list of codes housed in the dictionary "**Codes.dic"** in the "**dic-utils"** folder of the active language.

This function allows us to verify the spelling of entries in a specialized dictionary, as well as the codes used (ex. "ADJ" rather than "A").

The file "**def.lst"** used by default to check spelling of French dictionaries contains the only dictionary "**Delafm.bin"**. Here is the list of codes that are found in the "**Codes.dic"** dictionary:

```
ADV,.CAT
CONJC,.CAT
CONJS,.CAT
DET,.CAT
fp,.FLEX
fs,.FLEX
Hum,.SEM
mp,.FLEX
ms,.FLEX
N,.CAT
PREP,.CAT
PRO,.CAT
```

The list of dictionaries "**def.lst"** can be modified, for example is the user adds lexical entries which contain forms currently absent from the Delafm dictionary (numerous compound words have constituents which themselves are proper nouns, for example *algèbre de Boole*); similarly, the dictionary of codes "**Codes.dic"** must be edited if the user adds codes to the system (for example **+Finance** to describe terms of a financial nature).

Apply FST Format Description

This function allows us to verify the format of the dictionary files in a manner as precise as we desire (contrary to the "Fast, Minimal Check" option), thanks to description of the desired format using an INTEX grammar. The graphs used to check the four types of dictionaries are stored in the folder "**dic-utils**" of the current language: "**Check Delas.grf"**, "**Check Delac.grf"**, "**Check Delaf.grf"** and "**Check Delacf.grf**". Here, for example, is the graph used to check the format of a DELAS dictionary of French:

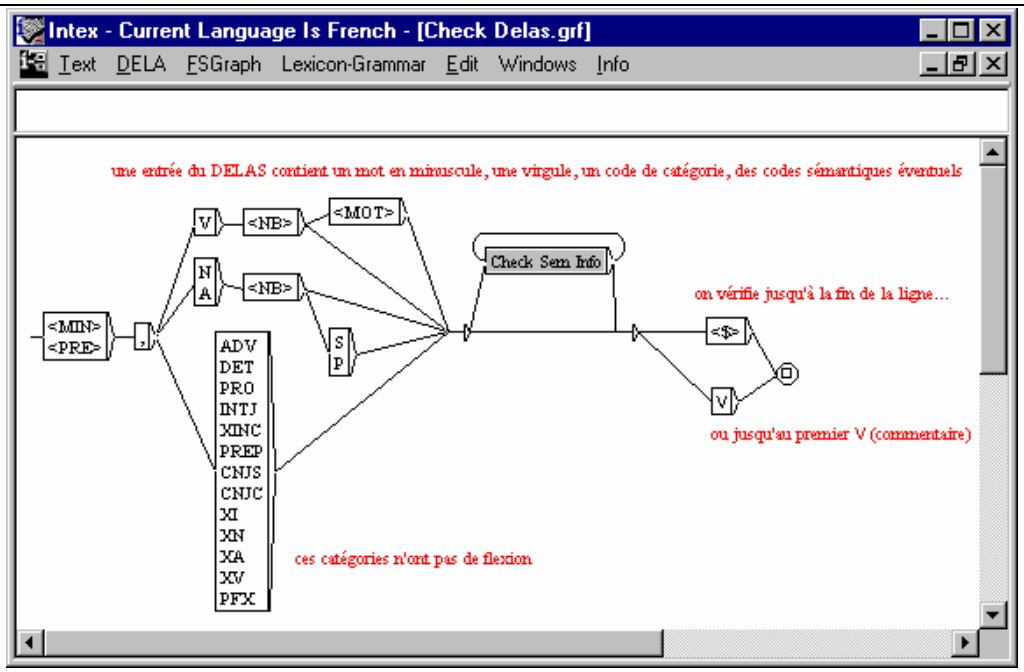


Figure 99. Formal verification of a DELAS

(the embedded graph Check Sem info checks the syntactico-semantic information introduced by the character "+"). The user can, of course, adapt these graphs to the dictionaries that they are using.

15.2. Automatic inflection of a DELA dictionary

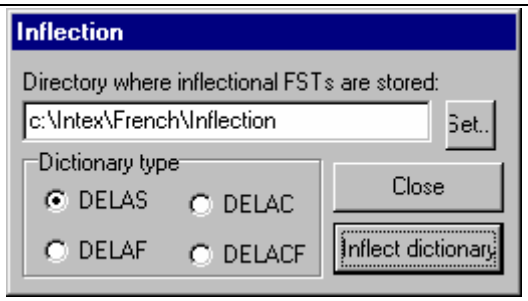


Figure 100. Automatic inflection

Automatic inflection has as it's goal, the construction of DELAF and DELACF dictionaries which will contain as entries, all forms which could appear in a given text. In principle, linguists construct DELAS and DELAC dictionaries, and the automatic inflection of these dictionaries produces the corresponding DELAF and DELACF dictionaries.

[Silberztein 1993], Blandine Courtois (at LADL), Cristina Mota (Univ. of Lisbon) and Agata Chrobot (Univ. de Tours) constructed programs that can generate DELACF dictionaries from DELACs.

The team of Annibale Elia (Univ. of Salerne) built a program for the inflection of a DELAF dictionary, which automatically adds suffixes to the adjectival forms (ex. "*-issimo"* in Italian) and generates all verbal forms combines with the verbal particles, for example, in Spanish: the form *digaselo* is obtained from the conjugated verbal form "*diga*" (*tell*) and the particles "*se*" and "*lo*" (*-it-to him/her*). See Chapter 12 for a description of the INTEX tokenizer and morphological parser.

Description of inflectional morphology

The folder "**Inflection"** contains a description of the inflectional morphology of the active language. This description will present itself in the form of a list of transducers, obtained either from graphs or from rational expressions. Each transducer describes how to obtain the total number of inflected forms, which correspond to a given word (the lemma). All the words which can be inflected in the same manner are associated with the same inflectional transducer.

The name of the transducer is the exact morpho-syntactic code that is associated with each DELAS entry. For example, for the following DELAS entry:

```
cousin,N32+Hum
```

there must exist, in the "**Inflection"** folder, a transducer "**N32.fst"** which generates the four forms *cousin, cousine*, *cousins* and *cousines*. All of the nouns inflected in this manner (ex. *ami*, *client*, *souverain*, etc.) are associated with the code **N32** in the DELAS dictionary.

Let's begin with a simple case; in general, **grammatical words** do not take inflection in French. This case is represented by transducers, which have, as an entry, an empty suffix (<E>), and produce therefore nothing. Here for example, the graph **PREP** is associated to the prepositions of the French DELAS:

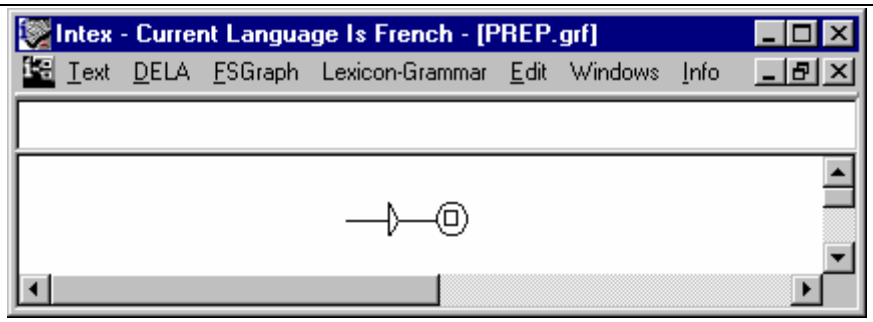


Figure 101. The inflectional transducer for PREP

French **Nouns and adjectives** can be inflected according to gender and number. This case involves transducers that contain as entries, the suffixes that must be added to

the lemma in order to obtain each form, followed by the corresponding inflectional codes. For example, here is the graph **N32** associated to nouns such as *cousin*:

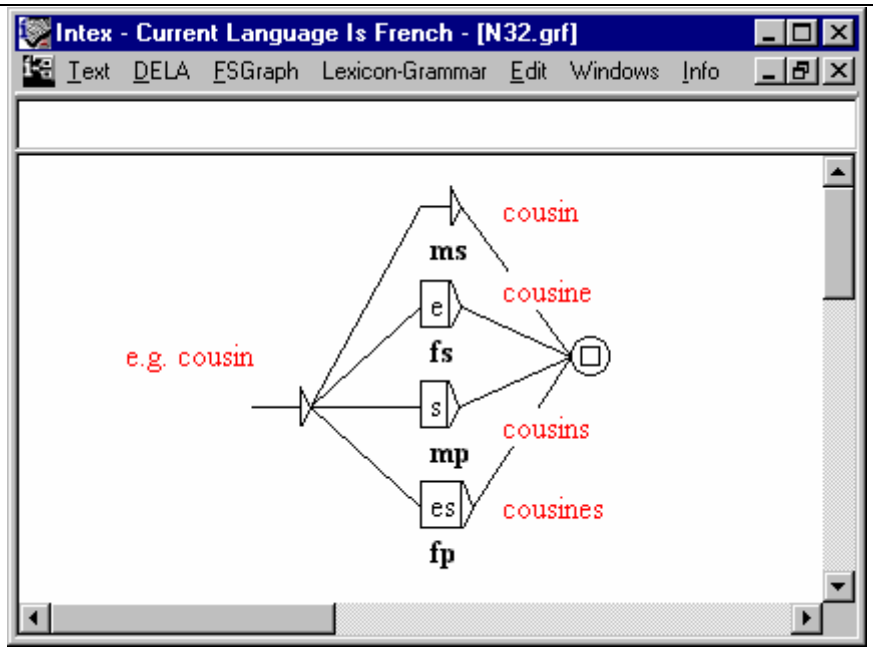


Figure 102. The transducer for the inflection of N32

This transducer is used in the following manner:

-- (upper path) if we add the empty suffix to the DELAS entry, we producer the form "*cousin*" associated with the inflectional codes "**ms**" (masculine singular) ;
-- if we add the suffix "*e*" to the DELAS entry, we produce the form "*cousine*" associated with the inflectional codes "**fs**" (feminine singular);
-- if we add the suffix "*s*" to the DELAS entry, we produce the form "*cousins*" associated with the inflectional codes "**mp**" (masculine plural);
-- if we add the suffix "*es*" to the DELAS entry, we produce the form "*cousines*" associated with the inflectional codes "**fp**" (feminine plural).

From the following DELAS entry:

```
cousin,N32+Hum
```

INTEX automatically generates the following DELAF (**DELA** > **Inflect** > **Inflect Dictionary**):

```
cousin,cousin.N32:ms+Hum
cousine,cousin.N32:fs+Hum
cousins,cousin.N32:mp+Hum
cousines,cousin.N32:fp+Hum
```

The inflection of nouns and adjectives is treated identically in the French DELAS dictionary (cf. [Courtois, 1990]). Based on the graph "**N32.grf**" we compile two

identical transducers "**N32.fst**" and "**A32.fst**", that are stored in the "**Inflection**" folder.

The nouns (or adjectives) that have identical inflected forms can be represented in a number of ways. For example, here is the graph **N31** that represents the inflection of nouns and adjectives such as "*artiste*":

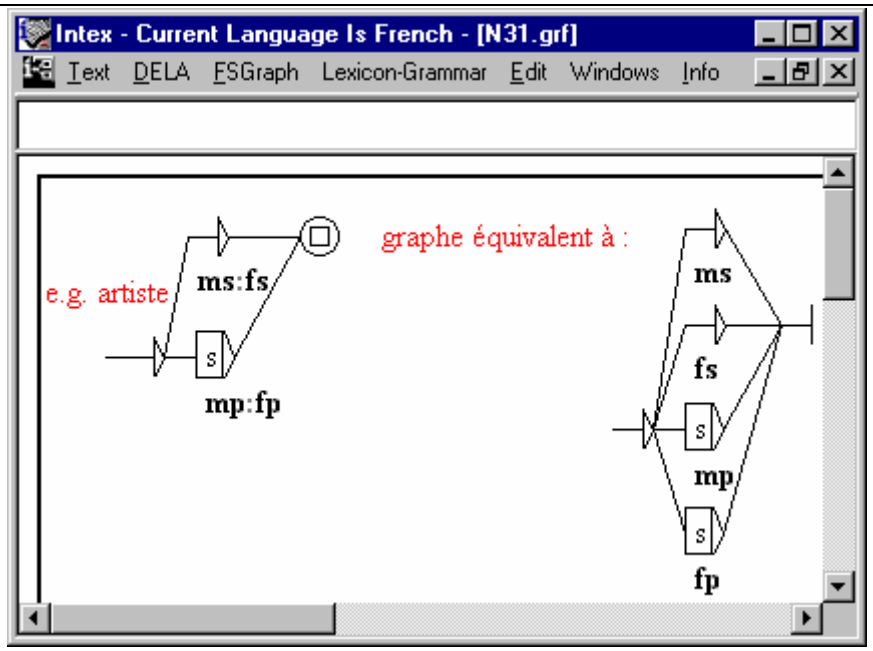


Figure 103. Inflectional transducer for N31

The graph on the left generated the following two entries in the DELAF:

```
artiste,artiste.N31:ms:fs
artistes,artiste.N31:mp:fp
```

while the graph on the right generated four entries:

```
artiste,artiste.N31:ms
artiste,artiste.N31:fs
artistes,artiste.N31:mp
artistes,artiste.N31:fp
```

The process of compressing the DELAF (**DELA > Compress into FST**) will take into account the inflectional information; the second graph (right) will automatically be replaced by the first (left).

The deletion operator

In French, numerous lemmas are not merely prefixes of their inflected forms; for example, the DELAS entry "*cheval*" is not a prefix of the form "*chevaux*". in order to obtain that form from the lemma, we must delete the last letter "*l*" of the lemma and then add the suffix "*ux*".

In INTEX, the last character is deleted using the operator "**L**" (*Left*):

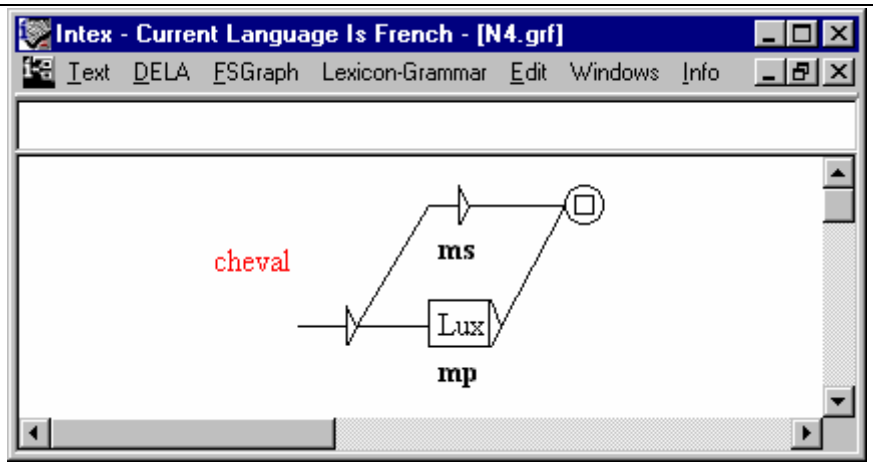


Figure 104. Transducer N4

The automatic inflection is accomplished in two steps:

```
cheval => chevalLux => chevaux
```

Thanks to this operator, we can represent all possible types of inflection, including those considered more "exotic"; for example:

```
recordman => recordmanLLLwoman => recordwoman
```

Verbal inflection is similar to nominal and adjectival inflection, except that we prefer to use rational expressions as opposed to graphs to construct the inflectional transducers. As an example, here is the rational expression "**V3.exp**" from which the transducer "**V3.fst**" is derived. This transducer conjugates the verbs of the first group (French 'ER' verbs):

```
L/P1s:P3s + Ls/P2s + 2ons/P1p + Lz/P2p + Lnt/P3p +
2ais/I1s + 2ais/I2s + 2ait/I3s + 2ions/I1p + 2iez/I2p + 2aient/I3p +
2ai/J1s + 2as/J2s + 2a/J3s + 2âmes/J1p + 2âtes/J2p + 2èrent/J3p +
ai/F1s + as/F2s + a/F3s + ons/F1p + ez/F2p + ont/F3p +
ais/C1s + ais/C2s + ait/C3s + ions/C1p + iez/C2p + aient/C3p +
L/S1s:S3s + Ls/S2s + 2ions/S1p + 2iez/S2p + Lnt/S3p +
2asse/T1s + 2asses/T2s + 2at/T3s + 2assions/T1p + 2assiez/T2p +
2assent/T3p +
L/Y2s + 2ons/Y1p + Lz/Y2p +
<E>/W + 2ant/G + 2é/Kms + 2ée/Kfs + 2és/Kmp + 2ées/Kfp
```

We can abbreviate a sequence of deletion operators by simply indicating the number of deletions: "*3*" is equivalent to "*LLL*". From the following DELAS entry:

```
aider,V3+t
```

the first three terms of the expression represent the following forms:

```
aide,aider.V3+t:P1s:P3s
aides,aider.V3+t:P2s
```

```
aidons,aider.V3+t:P1p
```

The “stack” operators

In theory, the concatenation of suffixes associated with the deletion operator allows us to represent any kind of inflection. For example, we obtain the conjugated form "*ont"* from the lemma *avoir* thanks to the suffix "**5ont"** (delete five letters, then add *ont*).

In practice however, the proposed system will lead to an artificial multiplication of the number of inflectional transducers. For example, consider the following four verbs and the suffixes to be added in order to produce the conjugated form associated with the third person singular in the present tense indicative:

```
acheter => 4ète => achète
  geler => 4èle => gèle
  mener => 4ène => mène
  semer => 4ème => sème
```

The conjugations of these four verbs are very similar, but require four different yet almost identical inflectional transducers, containing four different suffixes to produce the form. In languages like German or Greed, this would lead to adding hundreds of almost identical transducers.

To avoid that, INTEX contains two operators: "**R"** (*Right*) lets us skip a letter to the right in the lemma, an "**C"** (*Copy*) allows us to duplicate it. For example, from the four preceding infinitive verbs, we generate the present, third person singular form using the following command:

`LLLLRèCe` (ou de façon abrégée : `4RèCe`)

The successive operators can be visualized using a “stack”:

Lemma	Command	Stack	Result
acheter	L	*r*	*Achete*
achete	L	*er*	*Achet*
achet	L	*ter*	*Ache*
ache	L	*eter*	*Ach*
ach	R	*ter*	*Ach*
ach	è	*ter*	*Achè*

achè	C	*er*	*Achèt*
achèt	e	*er*	*Achète*

In the suffixes produced by the transducers, "L" corresponds to the stack operator PUSH, "R" corresponds to the stack operator POP, "C" corresponds to the stack operator POP&PRINT; each letter corresponds to the implicit operator PRINT.

Inflect Dictionary

To construct a DELAF, you must choose the type of dictionary (in this case DELAS), the folder in which the inflectional morphology is described – within the active language - (the folder "**Inflection"** is highlighted by default), then click on "**Inflect dictionary"**. The result, INTEX produces a DELAF type dictionary, and the list of entries, where applicable, that could not be inflected (due to errors).

The inflection of a DELAS dictionary is generally quite immediate: each operator takes a constant amount of time; the construction of each form takes an amount of time proportional to the length of the suffix indicated by the transducer; the inflection of each DELAS entry takes an amount of time proportional to the number of paths indicated by the transducer, in other words, the number of forms to be generated; the inflection of the DELAS takes an amount of time proportional to the size of the DELAF to be constructed.

Compress Dictionary into FST

When a DELAF (or DELACF) dictionary is constructed, it is immediately useable and can be installed as a "`.dic`" file in the **Delaf** (or **Delacf**) folder of the active language.

But in general these files are of an enormous size; for example the French DELAF, which contains roughly 700,000 forms, is over 30 Megabytes; the Russian DELAF, which contains roughly 3.5 million forms, is over 100 Megabytes in size. It is therefore advantageous to convert these dictionaries into transducers, of which the size is typically in the order of only several megabytes.

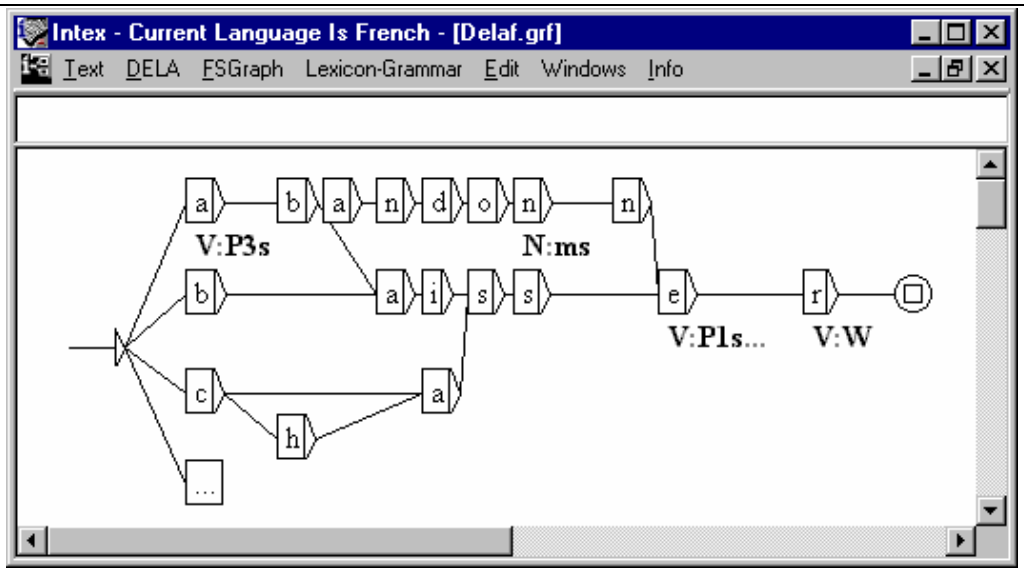


Figure 105. Excerpt from a dictionary represented as a transducer

(All of the nodes that contain an operation are treated as terminal nodes). This transducer recognizes the form *a*, the noun *abandon*, their infinitive verbs *abandonner, abaisser, baisser, casser* and *chasser*, and the conjugated forms *abandonne, abaisse, baisse, casse* and *chasse*.

This transducer is **deterministic** since the prefixes of all forms are factored into the equation. For example, even if several tens of thousands of forms begin with an "a", the node "a" is written but one single time in the transducer (whereas this letter would be written several tens of thousands of times in the dictionary);

This transducer is **minimalist** in that the suffixes of all forms are also factored in. For example, even if several tens of thousands of forms end in "*er*", and which correspond to the information "*Infinitive verb*", the suffix, as well as the corresponding information are only written one single time in the transducer (as opposed to several tens of thousands of times in the dictionary).

INTEX automatically constructs the equivalent transducer for each DELAF or DELACF dictionary (**DELA > Compress into FST**). The transducer is represented by two files: a "`.bin`" file, which contains the transducer as such, and an "`.inf`" file, which contains the vocabulary of the transducer, in other words, the sum total of the lexical information found in the dictionary. The "`.inf`" file can be edited, which allows us to replace codes at will (for example, if we want to replace the code "**A**" with the code "**ADJ**").

Chapter 16. LEXICON-GRAMMARS OF FROZEN EXPRESSIONS

The transducer for frozen expressions that we described in the chapter on "Lexical Resources" was constructed manually. But [Gross 1993] has inventoried more than 30,000 frozen expressions in French, housed in the tables **Cxx** of lexicon-grammar. The frozen expressions that make up a single table all share the same basic syntactic structure.

INTEX allows us to automate the construction of a transducer for each table. To do that, the following is required:

(1) A lexicon-grammar table;
(2) A master graph that formalizes the properties described in the table.

INTEX can then establish the correspondence between the properties of each entry and the paths of the related master graph; the result is a transducer with the same function and form as a transducer that would've been generated manually, but that would represent several hundred expressions.

16.1. Lexicon-grammar tables

In order for INTEX to be able to use it, the lexicon-grammar table must, of course be input into the machine. The most natural method of doing this is to use a spreadsheet application such as Microsoft Excel: each entry is described in a single row; the

properties are entered in columns; a cell in the spreadsheet (the intersection of a row and a column) contains either text or a sign "+" or "-".

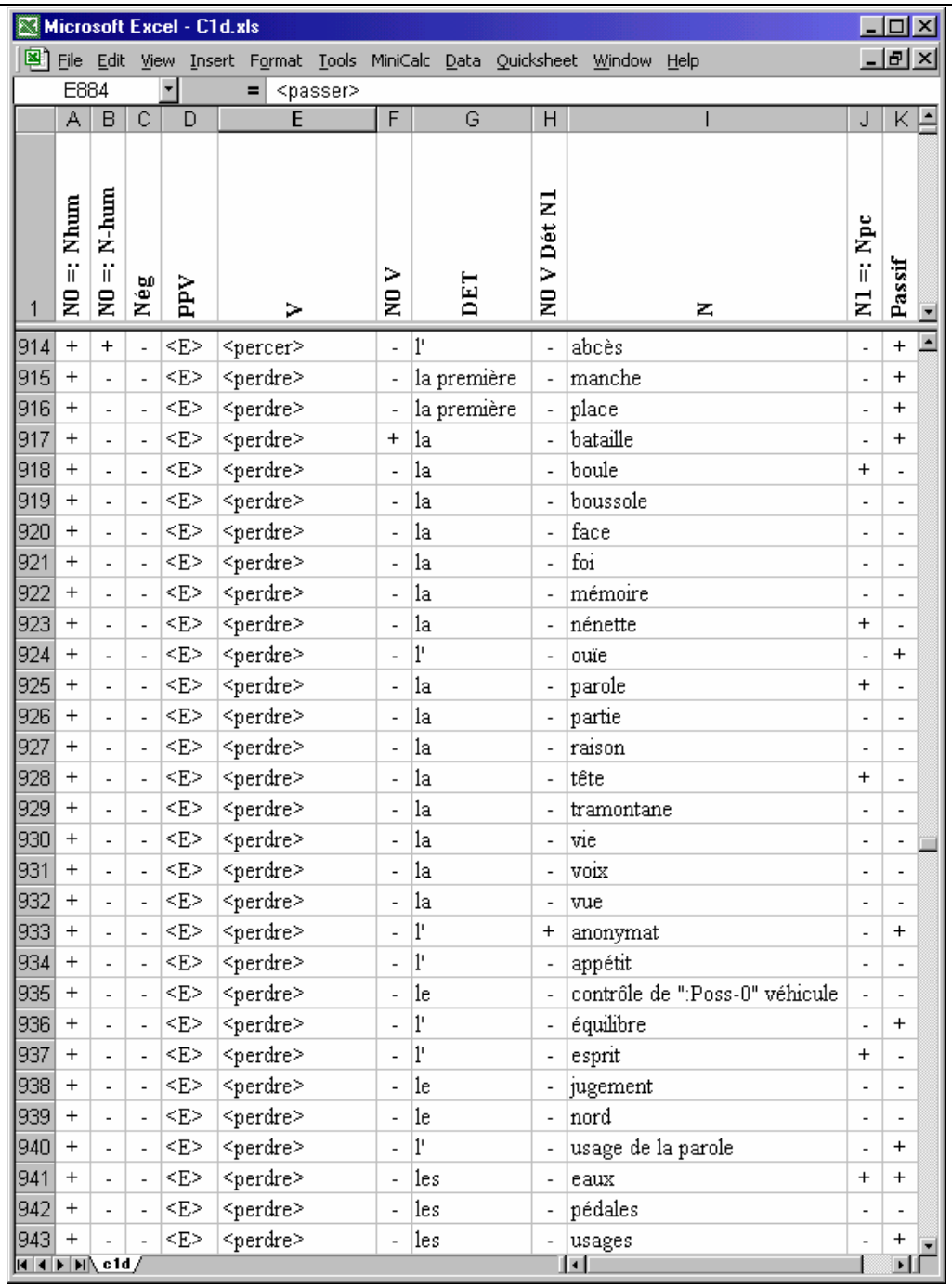
The format of the lexicon-grammar table must respect certain constraints, natural constraints for those familiar with lexicon-grammars:

1. The first line of the column contains the name of each zone of text or property (the "title" of each column); there must be as many titles as there are columns;

2. The table entries begin on the second line; each entry of the lexicon-grammar is described in a single row at the most; there cannot be an empty row;

3. A text cell contains a sequence that be comprised of constants (ex. "la raison"), lexical symbols (ex. "<perdre>", "<aller:P>" or "<PREP>") and references to graphs (ex. ":Dnum") ; the symbol "<E>" must be used to represent an empty string;

4. A property cell will exclusively contain the characters "+" or "-" (no character "?"); the columns must be homogenous: a property column cannot contain any text; likewise, a text column cannot contain the characters "+" or "-".



Microsoft Excel - C1d.xls

File Edit View Insert Format Tools MiniCalc Data Quicksheet Window Help

E884 = <passer>

1	A N0 =: Nhum	B N0 =: N-hum	C Nég	D PPV	E V	F N0 V	G DET	H N0 V Dét N1	I N	J N1 =: Npc	K Passif
914	+	+	-	<E>	<percer>	-	l'	-	abcès	-	+
915	+	-	-	<E>	<perdre>	-	la première	-	manche	-	+
916	+	-	-	<E>	<perdre>	-	la première	-	place	-	+
917	+	-	-	<E>	<perdre>	+	la	-	bataille	-	+
918	+	-	-	<E>	<perdre>	-	la	-	boule	+	-
919	+	-	-	<E>	<perdre>	-	la	-	boussole	-	-
920	+	-	-	<E>	<perdre>	-	la	-	face	-	-
921	+	-	-	<E>	<perdre>	-	la	-	foi	-	-
922	+	-	-	<E>	<perdre>	-	la	-	mémoire	-	-
923	+	-	-	<E>	<perdre>	-	la	-	nénette	+	-
924	+	-	-	<E>	<perdre>	-	l'	-	ouïe	-	+
925	+	-	-	<E>	<perdre>	-	la	-	parole	+	-
926	+	-	-	<E>	<perdre>	-	la	-	partie	-	-
927	+	-	-	<E>	<perdre>	-	la	-	raison	-	-
928	+	-	-	<E>	<perdre>	-	la	-	tête	+	-
929	+	-	-	<E>	<perdre>	-	la	-	tramontane	-	-
930	+	-	-	<E>	<perdre>	-	la	-	vie	-	-
931	+	-	-	<E>	<perdre>	-	la	-	voix	-	-
932	+	-	-	<E>	<perdre>	-	la	-	vue	-	-
933	+	-	-	<E>	<perdre>	-	l'	+	anonymat	-	+
934	+	-	-	<E>	<perdre>	-	l'	-	appétit	-	-
935	+	-	-	<E>	<perdre>	-	le	-	contrôle de ":Poss-0" véhicule	-	-
936	+	-	-	<E>	<perdre>	-	l'	-	équilibre	-	+
937	+	-	-	<E>	<perdre>	-	l'	-	esprit	+	-
938	+	-	-	<E>	<perdre>	-	le	-	jugement	-	-
939	+	-	-	<E>	<perdre>	-	le	-	nord	-	-
940	+	-	-	<E>	<perdre>	-	l'	-	usage de la parole	-	+
941	+	-	-	<E>	<perdre>	-	les	-	eaux	+	+
942	+	-	-	<E>	<perdre>	-	les	-	pédales	-	-
943	+	-	-	<E>	<perdre>	-	les	-	usages	-	+

c1d

Figure 106. Excerpt of the table **C1d**

The preceding excerpt contains all the frozen expressions of the table **C1d** which houses the verb *perdre*. Table **C1d** describes more than 1,500 frozen expressions of which the structure is: $\mathbf{N_0\ V\ N_1}$, where $\mathbf{N_0}$ represents the subject, **V** the verb and $\mathbf{N_1}$ the frozen element, the direct object..

16.2. Preparing the table

Here we will mention the modifications that must be made to the published tables in order for them to be processed by INTEX:

-- In the initial tables, there are no indications to distinguish between constant forms (e.g. *raison*) and forms that can be inflected (e.g. *perdre*). We must therefore replace each inflectionable form with the corresponding lexical symbol (e.g. `<perdre>` or `<voir:K>`) or by a reference to a mini-graph (for example. The graph Poss recognizes the possessive determiners *mon ton son*; the graph se recognizes the forms *me, te, se, nous, vous, m', t'* and *s'*) ;

-- The forms *de, le* and *la* are represented in their normalized form in the initial tables; we must elide them where necessary; similarly, the sequences *à le, à les, de le* and *de les* must be contracted where necessary. In the tables which incorporate free zones, we've replaced the prepositions and determiners bordering on the free zones *à, de* and *le* by mini-graphs that allow us to acknowledge the phenomena of contraction and elision (the graph à recognizes the three forms *à, au, aux*; the graph de recognizes the four forms *d', de, des, du*; the graph LE recognizes the four forms *le, la, les, l'*);

-- The presence of either a preverbal particle or a manditory negation (e.g. *Luc se creuse la cervelle*; *Luc n'a pas inventé la poudre*) was identified within the text of column **V**; we must extract this information from the text zone and add the corresponding property or text columns;

-- Classifiers and references to more or less limited groups of nouns (e.g. `"Norgane"`, `"Poss-0"`) must be replaced by references to embedded graphs (e.g. the graphs Poss-0 and Norgane) ;

-- in the initial tables, the quotation marks are used to cite examples (e.g. *fêter la "Saint-Léa"*); we've done away with these quotation marks. Note that these entries, given as examples, are always coupled with a more general entry (e.g. *fêter la ":Saint-Prénom"*); we could therefore eventually eliminate them.

When the table is in the correct format, it is necessary to save it in "text" mode; the zone separator is the tabulation character.

16.3. The master graph

The master graph of a lexicon-grammar table is used to interpret the properties described in the table; [E. Roche, 1993] described the first attempt to automatically build finite automation from a table and a master graph.

Each entry of the table will automatically and eventually be confronted by the master graph; this process results in the construction of a transducer identical to that which we would've constructed "by hand" for the expression represented by this row.

In INTEX, the lexical graphs that we can create to describe frozen expressions and the master graphs that describe the tables of frozen expressions are very similar. The only difference is that the master graph contains references to the cells. Hence, the reference "**@A**" represents the contents of the cell found at the intersection of the current row and the column "**A**" of the table, "**@B**" represents the contents of the cell at the intersection of the current row and the column "**B**", etc. (**A** being the first column, **B** the second, etc. this convention is used by the majority of spreadsheet applications).

For example, the graph representing the expression *perdre la raison* could be generalized by the following master graph:

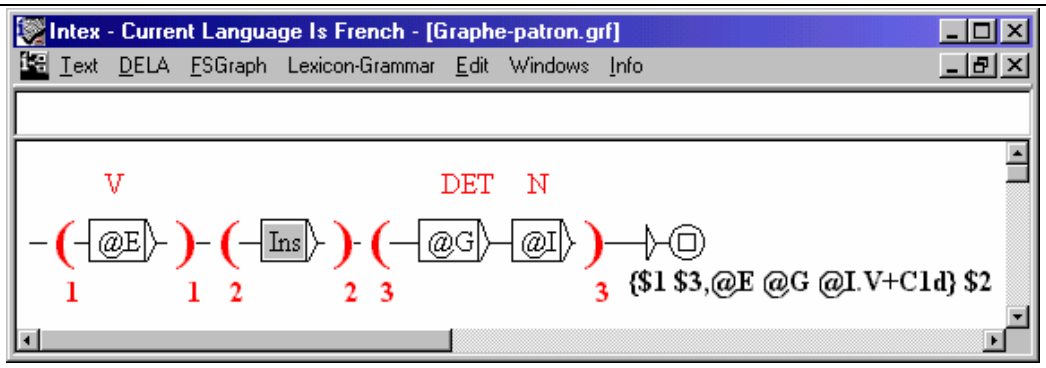


Figure 107. Example of a master graph

Note that, in the preceding figure, the column "**E**" of the table **C1d** represents the verb (e.g. `<perdre>`), "**G**" the determiner (e.g. `la`) and "**I**" the noun (e.g. `raison`). The variables are presented at the *outset* of the transducer to identify the constituents of the expressions in the text, as well as at the end of the transducer in order to lemmatize the recognized expressions. For example, the tag produced by the preceding transducer will contain, as a lemma, the text contained within the cells **E**, **G** and **I** of the table.

If indeed we replace the references to cells by their content, we obtain a graph practically identical to the graph used to tag the expression *perdre la raison* and its variants (described previously). The only difference is that in the preceding graphs, all of the variants were associated to the lemma *perdre la raison*; the preceding graph

lemmatizes each variant without linking it to the synonymous form (because the synonymous variants are not indicated in the table).

The preceding master graph doesn't take into account the following properties which are described in the table: **PPV** (mandatory preverbal particle), **N0 V** (optional object), **N0 V Dét N1** (free determiner) and **Passif** (the expression can be used in the passive voice). The following master graph is more comprehensive:

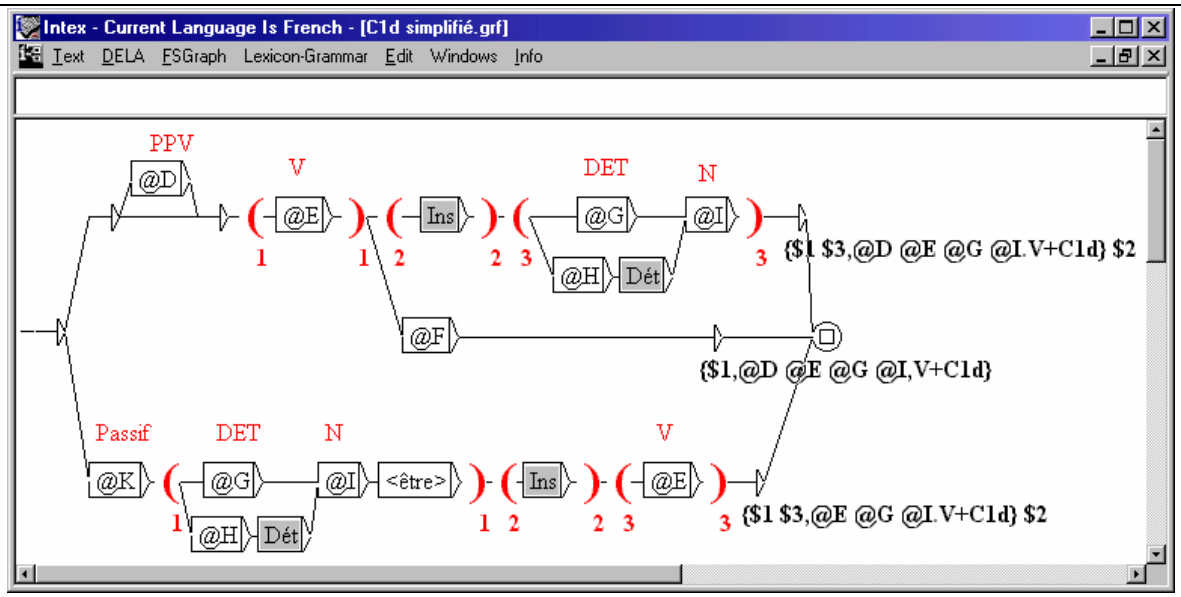


Figure 108. Simplified master graph

The properties @F (**N0 V**), @H (**N0 V Dét N1**) and @K (**Passif**) act as "doors" when it comes to the compilation of the graph: the paths, in the graph, that enter these doors are valid as long as the value of the variables in question is "+", and invalid when the value of the variables is "-".

For example, the expression *perdre la bataille* allows the property **N0 V**, which signifies that the expression can be abbreviated to *perdre*. Thanks to the middle path in the preceding master graph, INTEX recognizes and lemmatizes the abbreviated expression:

```
Luc a perdu => Luc a {perdu,perdre la bataille.V+C1d}
```

The tables are natural tools for describing such erasures, and the equivalent transducers can automatically render explicit the erased constituents of the texts.

16.4. Compound tenses

In the sentence:

```
Luc a perdu soudain la raison
```

the expression *perdre la raison* is conjugated in the “passé composé”. The ideal representation is the following:

```
Luc {a perdu la raison,perdre la raison.V} soudain
```

which comes back to saying that the form *a* is not an autonomous linguistic unit. Note that this lemmatization should, in principle, also be performed on free verbs, like for example in the following sentence:

```
Luc {a mangé,manger.V} une pomme
```

A priori, it would be somewhat natural to seek to resolve this problem in the same manner both for frozen expressions as well as for free verbs, in an individual syntactic component, independent of, et after any lexical analysis. After all, the more general problem of auxiliary verb usages, modal and transparent, is a syntactical problem (cf. M. Gross’ discussion on the web site, concerning the lemmatization of English verbs).

Compound tenses, however, represent a stumbling block to the recognition of frozen expressions. For example, the transducer obtained thanks to the preceding master graph does not recognize the frozen expressions in the following three sentences:

(se casser la figure) Luc et Marie **se** *sont* **cassé la figure**
(en faire le moins possible) Paul **en** *a toujours* **fait le moins possible**
(sauver la situation) **La situation** *a été soudain* **sauvée** *grâce à l’arrivée de Luc*

since the preverbal particles *se* and *en* are "too far" from the verb. It is therefore necessary to introducer at least a summary description of the compound tenses in the master graph of table **C1d**.

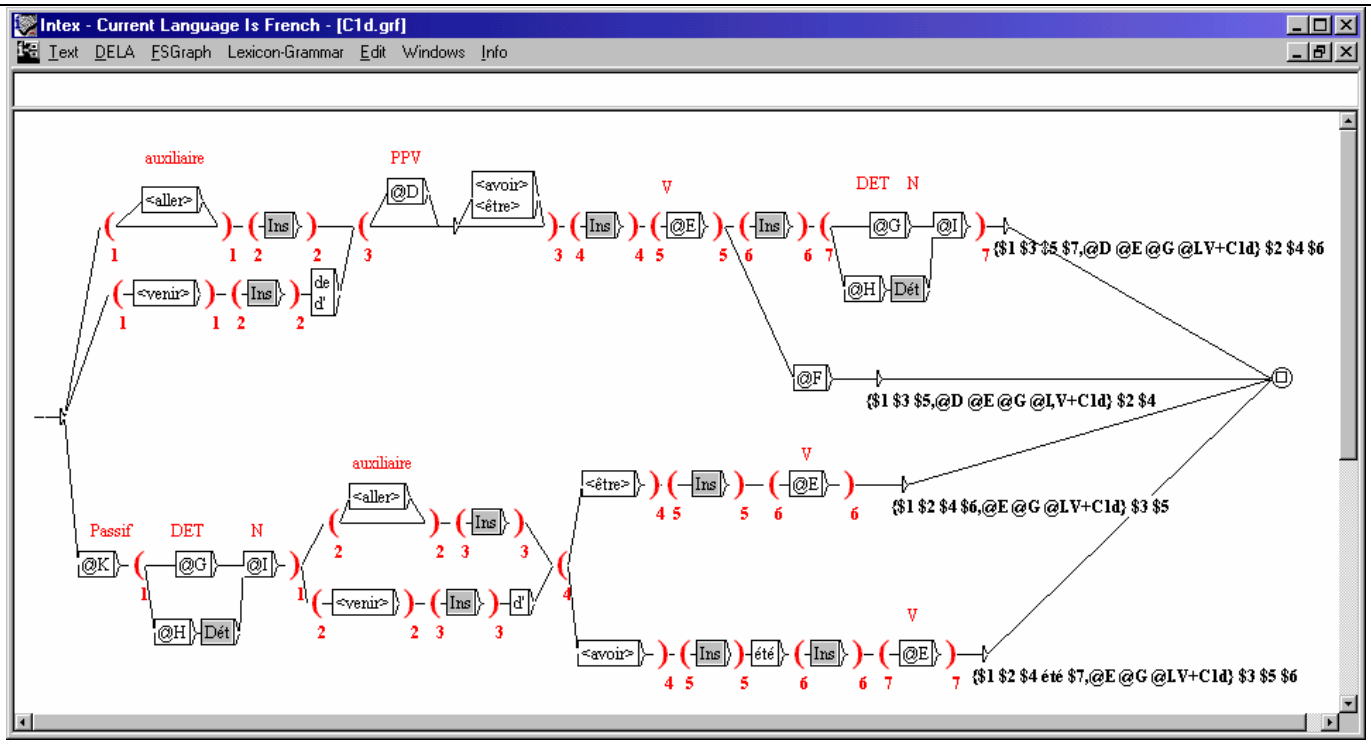


Figure 109. *Master graph of table* **C1d**

The master graph of table **C1d** takes into account the four auxiliaries *aller*, *venir de*, *être* and *avoir*.

16.5. Automatic compilation of the table

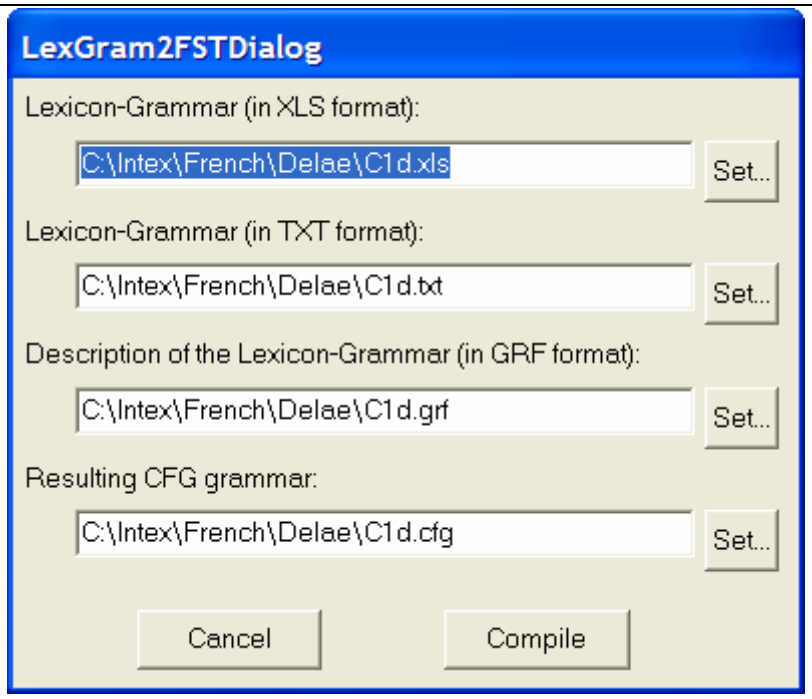


Figure 110. Compilation of a CFG grammar equivalent to a table

Using a lexicon-grammar table (for example **C1d.xls**); we can construct the file in text format, in which the fields are delimited by tabulation characters (as in **C1d.txt**). We then design the master graph (e.g. **C1d.grf**). The INTEX command **Lexicon-Grammar > Compile** is then used to build the equivalent graph (below, **C1d.cfg**).

Figure 111. Compilation of the table **C1d**

If all the properties of the table C1d have the value "+", each entry of the table will be associated with the 84 transformed sentences.

The equivalent transducer to the table **C1d** has a respectable size (cf. below): 1,662 expressions are described in the table; 50,742 phrases are effectively represented in the table, the resulting transducer contains 200,251 states.

The resulting grammar must be stored in the **Delae** folder of the current language in order to be used by INTEX's lexical module. Note that the .cfg files can also be used by the programs in the "Locate Pattern" module.

16.6. Perspectives

Some functionalities of program recognizing frozen expressions are as yet missing, as such, INTEX cannot yet include everything that might be needed for syntactic analysis.

Computing lexical information

We remind you that the phrase "*Luc perd la raison*" is actually tagged in the following manner:

```
Luc {perd la raison,perdre la raison.V+C1d}
```

INTEX does not contain a mechanism allowing it to recopy the lexical information associated with a given constituent, to the entire expression. For example, the verbal form *perd* is conjugated in the third person singular, in the present indicative, i.e. "`P3s`": we would like to be able to automatically recopy this information into the tag associated with the entire expression, which would yield the following tag:

```
{perd la raison,perdre la raison.V+C1d:P3s}
```

(We are talking about the *inheritance of properties* in the domain of syntactic analyzers). The mechanism that would be need to be added to INTEX to be able to accomplish this type of operation is not simple: for example, the sentence *Luc a cassé sa pipe* would need to be tagged as follows:

```
Luc {a cassé sa pipe,casser sa pipe.V+C1g:Q3s}
```

where "`Q3s`" would signify "passé composé, third person singular"; note that the form *a* is tagged "`P3s`" (present indicative, third person singular) and the form *cassé* "`Kms`" (participe passé, masculine singular). We would therefore need to be able to *calculate* "`Q`" from "`P`" and perhaps "`K`", then recopy "3s" all from the inflectional information associated with the auxiliary.

More generally, we cannot recopy syntactico-semantic information from a constituent to the entire expression: for example, the form *cassé* is associated with the property "`+t`" (*transitive*), while the expression *cassé sa pipe* would probably have the property "`+i`" (*intransitive*) since it no longer awaits the apposition of a nominal group acting as the object.

In time, we may wish to be able to name the properties, for example to transfer the property of gender or number associated with the verb in an expression, without necessarily wishing to transfer it's tense or mode, or perhaps the distributional class of one of the constituents of the expression but not it's structural properties, etc.

At the present however, it's unclear whether we really need such a sophisticated mechanism: the syntactic and distributional properties of expressions can be more readily rendered explicit in the tables (which comes back to adding columns, the value of which would be constant, in the table) or in the master graph (we could treat the four compound tenses separately).

VIII. INTEX for Developers

Chapter 17 describes each of the 30+ INTEX commands that are called ‘behind the screen” by the INTEX graphical interface; these commands are standalone programs that can also be directly used from a command-line DOS, or UNIX/LINUX Shell environment. Chapter 18 describes the files, directories and file formats used by INTEX and these programs. Chapter 19 gives a non-exhaustive list of references.

Chapter 17. COMMAND-LINE USE OF INTEX

17.1. Set up environment variables

Before using the Windows INTEX programs directly from a DOS command-line, the complete INTEX package must have been installed with the Windows installation utility "**Setup.exe**".

The various LINUX & UNIX packages come in different versions, but are always stored in the file: **unix-intex.zip**. This file must be uncompressed, and its content's file structure must be preserved.

The INTEX programs access the following five environment variables:

INTEX: variable that contains the name of the *application directory*, i.e. the directory in which the INTEX application was installed; by default in Windows: **c:\Program files\Intex**;

INTEXPRV: variable that points to the *private directory* of the current user, i.e. the directory in which all the user's files, dictionaries and grammars are stored; by default in Windows: **c:\Intex**;

INTEXAPP: variable that points to the application directory in which the INTEX programs are located; by default in Windows: **c:\Program files\Intex\App** (the installation file "license" should be stored there too);

INTEXLNG: variable that points to the application current language directory; for instance in Windows: **c:\Program files\Intex\French**;

INTEXLNG0: variable that points to the user's current language directory; for instance in Windows: **c:\Intex\French**.

In general, one wants to add **INTEXAPP** to the value of the **PATH** system variable, so that all INTEX programs can be launched from any directory.

Here is a sample of the file "autoexec.bat" used to launch INTEX programs from a DOS/Windows environment:

```
@set INTEX=c:\Program files\Intex
@set INTEXAPP=c:\Program files\Intex\App
@set INTEXPRV=c:\Max\Intex
@set INTEXLNG=c:\Program files\Intex\English
@set INTEXLNG0=c:\Intex\English
@set PATH=%PATH%;c:\Program files\Intex\App
```

Here is a sample of the file "profile.ksh" used to launch INTEX programs from a Windows/Bourne Shell environment (I use cygwin):

```
export INTEX="/c/Program files/Intex"
export INTEXAPP="$INTEX/App"
export INTEXPRV="/c/Max/Intex"
export INTEXLNG="$INTEX/English"
export INTEXLNG0="$INTEXPRV/English"
export PATH="$PATH;$INTEXAPP"
```

Here is a sample of the file ".profile" used to launch INTEX programs from a UNIX/Bourne Shell environment:

```
export INTEX="/usr/bin/Intex"
export INTEXAPP="$INTEX/App"
export INTEXPRV="/users/msilberz/Intex"
export INTEXLNG="$INTEX/English"
export INTEXLNG0="$INTEXPRV/English"
export PATH="$PATH;$INTEXAPP"
```

Make sure you do not overlook the difference between the **private** and the **application** directories:

-- the application directory is where the INTEX application is installed; it contains all the INTEX programs and DLLs, as well as the original dictionaries and grammars

included in INTEX. This directory and all its sub-directories should be write-protected, so that no user could accidentally destroy or modify any of their files;

-- each INTEX user on a PC has his/her own private directory; in this directory, are stored all the user's data: texts, dictionaries, grammars as well as results of all the processes: indices, concordances, statistical data, tagged texts, etc.

For instance, a user may edit a dictionary or grammar that is included in the INTEX package; a copy of the file will be stored in his/her private directory. For any further process that involves this dictionary or grammar, it is the user's version that will be used. Thanks to this mechanism, any number of users can share the same workstation without fear of having their work modified by each other.

Users can set their private directory via the menu item **Info > Preferences**.

17.2. Directories and files used by INTEX

Most files used by INTEX are text files; file formats are described in the next section.

All commands launched via the INTEX graphical interface are displayed in the INTEX console (**Info > Console**), with their arguments. Generally, one can copy & paste from this console to a shell window to launch the corresponding programs.

The content of the console is also stored in the file **log.txt**, which is stored in the private directory. This file is reset everytime INTEX is launched.

Text directory

When using the INTEX programs directly from the command-line or a shell window (i.e. without the INTEX graphical interface), one can store input files, temporary files and results of programs anywhere.

In the following, I describe where the INTEX application (i.e. when using the graphical interface) stores all its files.

When using the INTEX application to process a particular text, INTEX associates this text file with a directory, in which it stores the information associated with this text: the index of the text, its vocabulary, the concordances, etc. The text directory is stored beside the text file; its name is computed from the text file name by replacing the dot (character ".") with an underline character ("_"). For instance, if the text file name is:

```
c:\MyIntex\English\Corpus\Foobar.txt
```

then the text directory is:

```
c:\MyIntex\English\Corpus\Foobar_txt
```

Note: by default, INTEX sets text directories as "hidden"; check the option "**Tools>Display: display hidden files and directories**" of the Windows Folder option window if you need to see this directory and its content.

Files and special directories

Most INTEX programs use several specific files and directories:

Alphabet : this file lists the letters of the current language; this file must exist and be stored in the private or application directory of the current language.

idx, **ida** : the index of the text; these two files must be stored in the text directory. Note that the programs `dicos`, `reconind` et `recorind` access these two fichiers, even though these two file names do not appear as explicit arguments in the console.

Warning: every time a user opens a text via the INTEX application graphical interface, INTEX checks that the last modification date of the text file is earlier than the creation date of the two index files **idx** and **ida**. If these files do not exist, or if the text was modified after their creation, INTEX destroys all the files stored in the text directory.

license: installation file of the application. This file is used to check that the installation key corresponds to the serial number of the hard disk "**C:**", and to decrypt the dictionary files "**.bin"** stored in the application directory. This file must exist and be stored in the INTEXAPP directory (by default, the sub-directory App of the installation directory), and its content must not be modified.

Delaf, **Delacf**, **Delae** : INTEX expects that these directories exist, and are stored in the current language directory, both in the application (INTEXLNG) and in the private (INTEXLNG0) INTEX directories. The programs `dicos`, `dicoc` et `dicoe` access these directories when the special character "~" is used to locate various lexical resources.

Results of the processing

Most of the INTEX programs produce comments, error messages and results, such as "**Loading grammar DET**", "**Cannot find grammar AdvTime**", or "**235 matching sequences**".

The INTEX application writes usually these messages in the file "**res.txt**" of the private directory (**INTEXPRV**).

17.3. Commands

The following commands correspond to the ".exe" files that are stored in the application directory **INTEXAPP**.

`concord LLength RLength Tag? Text Index Concordance`

This program builds a concordance (`Concordance`), given: a text file (`Text`), an index of all matching sequences (`Index`), a length (number of characters) for the left context (`LLength`) and from the right context (`RLength`).

Warning: the parameter `RLength` counts the number of characters of the right context, including the ones of the matching sequences. If this parameter is lower than the length of some matching sequences, they will be incomplete.

This particular behavior ensures the proper display and printing of the concordance; each concordance entry being written in one line. By using adequate context lengths and fonts (fixed-size fonts, such as "Courier"), one can get beautiful concordances.

Note: If the parameter `RLength` is 0 (zero), matching sequences are displayed in their full length, but no right context is produced. Therefore, the list of all matching sequences can be produced by setting the two parameters `LLength` and `RLength` to 0 (zero).

The parameter `Tag?` must be "yes" to display tags in the concordance, "no" otherwise.

`dic2fst AsciiDelaf FstDelaf`

compiles a DELAF/DELACF-type dictionary into a minimal deterministic transducer. The transducer is then stored in two files: the file `FstDelaf.bin` contains the automaton, i.e. the recognizer; the file `FstDelaf.inf` stores the output of the transducer, i.e. the information associated with each lexical entry.

`dicoc [cdl] Text RDir Stats RDic Foo RIndex C-Dic...`

looks up dictionaries of compound words. The first parameter is a character that characterizes the type of the input text:

-- "`c`": the text is a concordance; as a consequence, dicoc only processes the second column of each line (columns are delimited by a tab character);
-- "`d`": text units (generally, sentences) are delimited by the tag: `{S}`;

-- "l" : text units are lines (or paragraphs), i.e. delimited either by the character "NEW LINE", or by the sequence of two characters "CARRIAGE RETURN" - "NEW LINE".

The second parameter is the text file name in which dicoc looks for compound words; the third parameter is the directory name in which the four resulting files are to be stored:

-- Stats: file (stored in directory RDir) that contains the number of recognized compounds;

-- RDic: file (stored in RDir) that contains the list of all recognized compounds; this file is named "DLC" by the INTEX application (i.e. when using the graphical interface to lookup compounds in a text), and is the "compound word vocabulary" of the text.

-- Foo: any file name (this parameter is kept for compatibility reasons); before 4.2x, this file would store all unambiguous compound words. Since 4.20, compound words ambiguities are no longer processed by the lexical module; INTEX now processes all disambiguations in the Disambiguation module.

-- RIndex : file name (file is stored in RDir) that contains the index of all recognized compounds.

The next parameters C-Dic are file names for dictionaries (extension ".bin" or ".dic") or lexical transducers (extension ".fst") to be applied.

-- the special character "~" may be used; in that case it specifies the directory in which the lexical resource is to be found: the program looks for the file in the **Delacf** sub-directory of the current language directory, first in the private directory, then, if not found, in the application directory. For instance, "**~Nouns.bin**" refers to the dictionary "**Nouns.bin**" that is stored either in the private directory "**c:\My Intex\English\Delacf**", or in the application directory "**c:\Program files\Intex\English\Delacf**".

-- the last character of the file name (before the dot) refers to the priority of the lexical resource. If it is a "-", then the resource is applied first; if it is a "+", then it is applied last (i.e. only when the sequence has not matched any compound yet).

dicoe [cdl] Text RDir Stats RDic ResIndex DLF DLC FSTs...

recognizes frozen expressions in the text. The first parameter describes the type of the input text:

-- "c": the text is a concordance; as a consequence, dicoe only processes the second column of each line (columns are delimited by a tab character);
-- "d": text units (generally, sentences) are delimited by the tag: {S};

-- "`l`" : text units are lines (or paragraphs), i.e. delimited either by the character "NEW LINE", or by the sequence of two characters "CARRIAGE RETURN" - "NEW LINE".

The second parameter is the text file name in which frozen expressions are searched; the third parameter is the directory name in which the following three result files are to be stored:

-- `Stats`: file (stored in `RDir`) that contains the number of frozen expressions found in the text;
-- `RDic`: file (stored in `RDir`) that contains the list of all recognized frozen expressions; this file is named "DLE" by the INTEX application (i.e. when using the graphical interface to lookup frozen expressions in a text), and is the "frozen expression vocabulary" of the text.
-- `RIndex`: file (stored in `RDir`) that contains the index of all recognized frozen expressions.

The two next parameters `DLF` et `DLC` are the simple word and compound word vocabulary files of the text; these two files are usually constructed by `dicos` and `dicoc`. They are generally used in order to recognize frozen expressions; the special argument "`-`" may be used instead of a vocabulary file.

The next parameters FSTs are the file names of the frozen expressions grammars (extension "`.fst`") to be applied to the text. These ".fst" files are either "complete" finite-state transducers, or "incomplete transducers. (see below).

-- the special character "~" may be used; in that case it specifies the directory in which the lexical resource is to be found: the program looks for the file in the **Delae** sub-directory of the current language directory, first in the private directory, then, if not found, in the application directory. For instance, "**~C1D.fst**" refers to the frozen expression grammar "**C1D.fst**" that is stored either in the private directory "**c:\My Intex\English\Delae**", or in the application directory "**c:\Program files\Intex\English\Delae**".

-- the last character of the file name (before the dot) refers to the priority of the lexical resource. If it is a "-", then the resource is applied first; if it is a "+", then it is applied last(i.e. only when the sequence has not matched any expression yet).

These FSTs can be either:

-- "real" Finite-State transducers, i.e. fully autonomous, in general are compiled from one or more graphs with the command **FSGraph > Compile FST** or the program grf2fst.exe;

-- "Context-Free" transducers contain unsolved references to grammars that are stored either in the same directory as the FST, or in the sub-directory "**Graphs\Lib**" of the private or application INTEX directory, such as Ins or Date. These grammars are

usually built from a meta-graph and a lexicon-grammar table. References are resolved by the program `dicoe`.

`dicos [cdl] Text RDir RDic RUnknown Stats S-Dic...`

looks up dictionaries and lexical transducers for simple words. The first parameter is a character that characterizes the type of the input text:

-- "`c`": the text is a concordance; as a consequence, dicoc only processes the second column of each line (columns are delimited by a tab character);
-- "`d`": text units (generally, sentences) are delimited by the tag: `{S}`;
-- "`l`" : text units are lines (or paragraphs), i.e. delimited either by the character "NEW LINE", or by the sequence of two characters "CARRIAGE RETURN" - "NEW LINE".

The second parameter is the text file name in which dicos looks for simple words; the third parameter is the directory name in which the four resulting files are to be stored:

-- `RDic`: file (stored in `RDir`) that contains the list of all recognized simple words; this file is named "DLF" by the INTEX application (i.e. when using the graphical interface to lookup simple words in a text), and is the "simple word vocabulary" of the text.

-- `RUnknown`: file (stored in `RDir`) that contains all unknown simple word forms, i.e. all word forms of the text that have not been found in any of the dictionaries, or have not matched any of the morphological grammars.

-- `Stats`: file (stored in directory `RDir`) that contains the number of recognized simple words;

The next parameters `S-Dic` are file names for dictionaries (extension "`.bin`" or "`.dic`") or lexical or morphological transducers (extension "`.fst`") to be applied.

-- the special character "~" may be used; in that case it specifies the directory in which the lexical resource is to be found: the program looks for the file in the **Delaf** sub-directory of the current language directory, first in the private directory, then, if not found, in the application directory. For instance, "**~ProperNames.dic**" refers to the dictionary "**ProperNames.dic**" that is stored either in the private directory "**c:\My Intex\English\Delaf**", or in the application directory "**c:\Program files\Intex\English\Delaf**".

-- the last character of the file name (before the dot) refers to the priority of the lexical resource. If it is a "-", then the resource is applied first; if it is a "+", then it is applied last (i.e. if no lexical entry has been found for the word form).

`enrich Text Index Result`

This program takes the text file `Text` as an input, and a file that contains the index of a set of sequences `Index`; each of these sequences being associated with a replacement string. The program then creates the resulting text file `Result`. Generally, the index has been created by one of the FST recognition programs (`recon`, `reconind`, `recor` ou `recorind`). When using the INTEX graphical interface, modifications to be applied to the input text to produce the result are of two kinds: either matching sequences are replaced with the corresponding output of the transducer, or the output of the transducer is inserted in the text.

`etiqa S-Dic C-Dic E-Dic Text MFT-Text Stats`

`etiqa` is used to construct the series of FSTs that represent the lexical analysis of the text. Each text unit (usually, sentence) of the text is represented by exactly one FST in the multiple FST file `MFT-text`.

`S-Dic` is the vocabulary file that contains all the simple words of the text (usually built with the `dicos` program); `C-Dic` is the vocabulary file that contains all the compounds of the text (usually built with the `dicoc` program); `E-Dic` is the vocabulary file that contains all the frozen expressions of the text (usually built with the `dicoe` program); `Text` is the input text (it should be in the "`.snt`" format); the resulting series of FSTs is stored in `MFT-text`; `Stats` includes some information (number of states and transitions of the resulting FSTs).

`etiqc [cdl] Text TaggedText C-Dic...`

`etiqc` is used to recognize and tag all compounds of the text. The first parameter is:

-- "`c`": the text is a concordance; as a consequence, dicoc only processes the second column of each line (columns are delimited by a tab character);
-- "`d`": text units (generally, sentences) are delimited by the tag: `{S}`;
-- "`l`" : text units are lines (or paragraphs), i.e. delimited either by the character "NEW LINE", or by the sequence of two characters "CARRIAGE RETURN" - "NEW LINE".

The second parameter is the input text; The resulting (partially) tagged text is `TaggedText`.

The following parameters `C-Dic` are file names of dictionaries and lexical transducers for compounds (files "`.bin`", "`.dic`" or "`.fst`"):

-- the special character "~" may be used; in that case it specifies the directory in which the lexical resource is to be found: the program looks for the file in the **Delacf** sub-directory of the current language directory, first in the private directory, then, if not found, in the application directory. For instance, "**~Nouns.bin**" refers to the dictionary "**Nouns.bin**" that is stored either in the private directory

“**c:\My Intex\English\Delacf**”, or in the application directory “**c:\Program files\Intex\English\Delacf**”.

-- the last character of the file name (before the dot) refers to the priority of the lexical resource. If it is a “-”, then the resource is applied first; if it is a “+”, then it is applied last (i.e. only when the sequence has not matched any compound yet).

`etiqg [Ddl] [0-7] Text FST SDic CDic NACDic Result Stats`

The INTEX tagger. In INTEX, tagging a text means replacing all unambiguous and disambiguated simple word forms and compounds with the corresponding lexical entry. Lexical entries are written in the form of tags, between curly brackets “{“ and “}”.

The first parameter corresponds to the type of the input text:

-- “D”: the input is a dictionary, in which each entry is a text that needs to be tagged (e.g. a DELAC-type dictionary contains compound word sequences in the entry field; these entries need to be tagged in order to produce the corresponding DELACF-type dictionary);
-- “d”: the input is a text in the INTEX format: “`.snt`”;
-- “l”: the input is a text that is line delimited (NEWLINE or CARRIAGE-RETURN, NEWLINE).

The second parameter corresponds to the type or tagging to be performed, as well as the file format of the result:

-- “0”: lexical information are written between parentheses, after each word form (this format is best to tag dictionaries);
-- “1”: tags contain the whole lexical information;
-- “2”: idem; all recognized compounds are tagged (even potentially ambiguous ones);
-- “3”: tags contain only the lemma (this format is used to lemmatize texts);
-- “4”: idem; all recognized compounds are tagged (even potentially ambiguous ones);
-- “5”: tags contain only the lemma and the morpho-syntactic category;
-- “6”: idem; all recognized compounds are tagged (even potentially ambiguous ones);
-- “7”: the result is a regular expression, that can also represent ambiguities, between several simple word solutions, as well as between simple and compound words.

The fourth parameter `FST` is the local grammar used to disambiguate the text. When using the graphical interface of INTEX, this FST is usually the union of all the local grammars that are selected in the **Disamb** panel. The two next parameters are the vocabulary files of the text: `SDic` includes the simple words and `CDic` includes the compounds (they are usually created by `dicos` and `dicoc`). The next parameter `NACDic` is a dictionary that contains unambiguous compounds. The resulting tagged

text is `Result`; a few counts (number of disambiguated sequences, number of inconsistencies between the vocabulary files and the disambiguation grammar) are stored in the resulting file `Stats`.

`flexion FST-directory DELAS DELAF errors`

This program automatically inflects a DELAS-type dictionary into a DELAF-type dictionary. The first parameter is the directory in which all the inflectional transducers are stored (files ".fst"); the program produces a DELAF-type dictionary and a file that contains all the entries of the DELAS dictionary that could not be inflected.

`fst2txt [cdl] [gp] [fsr] FST Text SDic NACDic ACDic Res`

Application of a syntactic transducer to a text.

`[cdl]`: type of the text file ("c": concordance file, "d": ".snt" file delimited with "{S}" tags; "l": text file delimited by NEWLINE sequences);

`[gp]`: priority is given to the longest matches ("g") or shortest ones ("p");

`[fsr]`: the output of the grammar FST is inserted in the text ("f"); the output of the FST replaces the corresponding matching input ("s"); the output of the FST is ignored ("r");

`FST`: this is the syntactic grammar (.fst file) to be applied to the text; the grammar can represent a simple pattern, such as "a determiner followed by a noun", or be much more complex and represents a full part of a language (in that case, the FST has been generally compiled from a set of graphs).

`Text`: the input text (file ".txt" or ".snt");

`SDic` : the vocabulary file that contains all the simple words of the text;

`NACDic` : the vocabulary file that contains all the unambiguous compounds of the text;

`ACDic`: the vocabulary file that contains all the potentially ambiguous compounds of the text;

`Res`: the resulting text.

`genere Grammar Limit Format Language`

program that explores all the paths of a morphological or syntactic grammar in order to produce the corresponding language.

`Grammar`: this is either a graph ".grf" file, or a compiled transducer ".fst"); note that when exploring a ".grf" file, genere does not explore its embedded subgraphs.

`Limit`: if this paramter is 0 (zero), `genere` produces the set of all the possible sequences recognized by the grammar (i.e. the corresponding language). If the paramater is a positive number, it is interpreted as the maximum number of sequences to be produced; `genere` stops when this number has been reached; if this parameter is a negative number, it is interpreted as a time limit; `genere` stops when this number of seconds has been spent.

`Language`: the resulting file contains the sequences recognized by the grammar.

`Format`: there are four types of formats for the resulting file:

"1": input "=>" output; for instance: `tables => table.N:fs`
"2": DELAF-type format; for instance: `tables,table.N :fs`
"3": DELACF-type format; e.g.: `tables rondes,table ronde.N:fp`
"4": synchronized input/output; for instance:

```
la_<PRO> vole_<V>
```

(if the recognized input is "*la vole*" and the corresponding output is "<*PRO*> <*V*>").

`gr2fst GraphDir Graph Fst Determ?`

Compiles the corresponding finite-state transducer (file ".fst") from a given graph and all its embedded sub-graphs.

`GraphDir`: primary directory in which the program looks for the embedded graphs;

`Graph`: file name of the graph to be compiled;

`FST`: the resulting transducer;

`Determ?` "yes" if the transducer has to be deterministic; "no" otherwise.

There might be references to embedded graphs in the graph to be compiled, as well as in those graphs, etc. These references are either absolute file names, e.g. "c:\Intex\English\Graphs\Adverbs\Date\HourTime.grf", or relative file names, e.g. "HourTime". In the latter case, gr2fst looks for the corresponding ".fst", then ".grf" file in the following directories:

-- the primary directory `GraphDir`
-- the subdirectory "Graphs\Lib" included in the current language of the private directory, e.g. "c:\My Intex\English\Graphs\Lib"
-- the subdirectory "Graphs\Lib" included in the current language of the application directory, e.g. "c:\Program files\Intex\English\Graphs\Lib"

indexer [cdl] Text Keys Pos Stats TokensList CharsList

This program tokenizes, then creates the index of the text file `Text`. The index keys are tokens of four types:

-- sequences of letters (simple word forms);
-- digits (characters from “0” to “9”);
-- tags (in fact, any sequence of characters between curly brackets “{” and “}”;
-- delimiters (all other characters).

`[cdl]`: type of the text file (“c”: concordance file, “d”: “.snt” file delimited with “{S}” tags; “l”: text file delimited by NEWLINE sequences);

The resulting index is stored in two files: `Keys` contains the list of all the tokens, and `Pos` contains the positions in the text of all the occurrences of each index key. `TokensList` contains the list of the 100 most frequent tokens (with their frequency); `CharsList` contains the list of all the characters of the text file, with their frequency.

interg Grammar MFT-Text ResMFT-Text Stats

Program for the disambiguation of a text represented by FSTs.

`interg` applies the disambiguation local grammar `Grammar` to the text represented by the multiple transducer file `MFT-Text`, and computes the intersection between the grammar and the transducer of each sentence of the text. The resulting multiple transducer file is `ResMFT-Text`.

The grammar is a “`.fst`” file; when using the INTEX graphical interface, this FST is the union of all the selected disambiguation local grammars of the **Disamb** panel.

The multiple transducer `MFT-Text` is usually created by the program `etiqa`. It contains exactly one FST for each text unit (usually, sentence) of the text.

next2iso next-file iso-file

Program to convert NextStep or OpenStep text and dictionary files to Windows ANSI.

parse [WPS] [SLA] [limit] Grammar MFT-Text ResText Stats

The INTEX syntactic parser. Builds the tree that represents the structure of each recognized sentence (“normal” mode), or the derivation tree of the parsing process ‘”debug” mode).

`Grammar` usually is a Recursive Transition Network (RTN). The outputs of the grammar are labeled parentheses that are used to represent the structure of the

recognized sequences; generally, labeled parentheses are placed around meaningful phrases or modifiers.

Note that these parentheses produce a structure that can be largely independant from the structure of the grammar itself. If the grammar has no output, then it is equivalent to a Context-Free Grammar (CFG). If the graph of the grammar has no reference to embedded graphs, then it is equivalent to a Finite-State Grammar (FSG).

The input text `MFT-TEXT` contains a series of transducers (one for each sentence of the text). It is usually created by the programs `etiqa` and `interg`.

`[WPS]`: the grammar is to be applied to the whole sentences ("W"); to the prefix of each sentence ('P"), or the suffix of each sentence ("S").

Warning: if the parameter "W" is used, do not forget to include in the grammar the possible sentence delimiters, such as the final period "." or question mark. Otherwise, no sentence will be recognized...

`[SLA]`: the grammar produces only the longest matches ("L"), only the shortest matches ("S"), or all the matches ("A").

The resulting file `ResText` contains the list of all the matches, in which the structure information (the labeled parentheses produced by the grammar) have been inserted. Each matching sequence can then be displayed graphically (as a tree) by the INTEX application.

`re2fst RegExp FST`

constructs the finite state automaton file `FST` from a regular expression file `RegExp`.

`recondic FST Dictionary IncorrectEntries`

Applies an automaton to a dictionary text file, and produces the list of all the dictionaries entries that do **not** match. The grammar is in fact an finite state transducer (".fst" file), in which the outputs are simply ignored.

Usually, grammars represent the correct format of the entries of a dictionary; `recondic` is used to look for input errors and typos in text dictionaries ".dic" files. Several examples of such grammars are included in the INTEX package (in the "Dic-utils" subdirectory of each language).

The following four programs: `recon`, `reconind`, `recor` and `recorind` are variants of the same functionality: to apply a grammar to a text; the result is the index of all matching sequences of the text. `recon` and `reconind` apply a Finite-State grammar (more precisely, a ".fst" file) to a text; `recor` and `recorind` apply a Context-Free grammar (a ".grf" graph file that optionnally includes references to embedded graphs) to the text. `recon` and `recor` apply the grammar to the text

itself, whereas `reconind` and `recorind` use the index of the text to accelerate the process.

`recon [cdl] [gpt] [fsr] Limit FST Text S-Dic NAC-Dic AC-Dic ResIndex Stats`

Applies the finite-state grammar `FST` (in the form of a ".fst" file) to the text file `Text`. The grammar `FST` is a finite-state transducer (file ".fst"); usually, this file was compiled from a regular expression (by program `re2fst`) or from one or more graphs (by program `gr2fst`).

`[cdl]`: type of the text file: "c": concordance file (file format ".con", usually produced by the `concord` program); "d": text in which text units are delimited by the tag "{S}" (file format ".snt", usually produced by the program `fst2txt`); "l": raw Windows ANSI text (file format ".txt", processed line by line);

`[gpt]`: result includes only the longest matching sequences ("g"), only the shortest ones ("p"), or all matching sequences ("t");

`[fsr]`: the program inserts the output of the grammar in the text ("f"); the program replaces all matching sequences with the output of the grammar ("s"); the program ignores the output of the grammar ("r");

`Limit`: the program stops after `Limit` number of matches. If `Limit` = 0 (zero), then there is no limit;

When applying the grammar to the text, the following vocabulary files may be used:

`S-Dic`: dictionary that contains all the simple words of the text;
`NAC-Dic`: dictionary that contains all the unambiguous compound words of the text;
`AC-Dic`: dictionary that contains all the (potentially ambiguous) compounds words of the text;

The result of the processing is the index of all matching sequences (plus eventually the corresponding output), stored in file `ResIndex`.

`reconind [cdl] [gpt] [fsr] Limit FST Text S-Dic NAC-Dic AC-Dic ResIndex Stats`

Applies the finite-state grammar `FST` (in the form of a ".fst" file) to the text file `Text`. The index of the text must be stored in the two special files "**idx**" (list of the tokens of the text) and "**ida**" (the positions of the occurrences of the tokens in the text); these two files must be located in the directory of the text, according to the INTEX application default. For instance, if the text is stored in the file:

```
c:\Intex\English\Corpus\foobar.snt
```

then the directory of the text is:

```
c:\Intex\English\Corpus\foobar_snt
```

and the two following index files must exist:

```
c:\Intex\English\Corpus\foobar_snt\idx
c:\Intex\English\Corpus\foobar_snt\ida
```

The grammar `FST` is a finite-state transducer (file ".fst"); usually, this file was compiled from a regular expression (by program `re2fst`) or from one or more graphs (by program `gr2fst`).

`[cdl]`: type of the text file: "c": concordance file (file format ".con", usually produced by the `concord` program); "d": text in which text units are delimited by the tag "{S}" (file format ".snt", usually produced by the program `fst2txt`); "l": raw Windows ANSI text (file format ".txt", processed line by line);

`[gpt]`: result includes only the longest matching sequences ("g"), only the shortest ones ("p"), or all matching sequences ("t");

`[fsr]`: the program inserts the output of the grammar in the text ("f"); the program replaces all matching sequences with the output of the grammar ("s"); the program ignores the output of the grammar ("r");

`Limit`: the program stops after `Limit` number of matches. If `Limit` = 0 (zero), then there is no limit;

When applying the grammar to the text, the following vocabulary files may be used:

`S-Dic`: dictionary that contains all the simple words of the text;
`NAC-Dic`: dictionary that contains all the unambiguous compound words of the text;
`AC-Dic`: dictionary that contains all the (potentially ambiguous) compounds words of the text;

The result of the processing is the index of all matching sequences (plus eventually the corresponding output), stored in file `ResIndex`.

recor [cdl] [gpt] [fsr] Limit GRF Text S-Dictionary NAC-Dictionary AC-Dictionary Index Stats

Applies the grammar `GRF` to the text file `Text`. The grammar `GRF` is a Recursive Transition Network represented by a graph file (file ".grf" or ".cfg"), and may include references to embedded graphs. If the grammar has no output (or if the parameter "r" is used), the series of graphs is equivalent to a Context-Free Grammar; if the graph contains no reference to embedded graphs, it is equivalent to a Finite-State Grammar.

`[cdl]`: type of the text file: “c”: concordance file (file format “.con”, usually produced by the `concord` program); “d”: text in which text units are delimited by the tag “{S}” (file format “.snt”, usually produced by the program `fst2txt`); “l”: raw Windows ANSI text (file format “.txt”, processed line by line);

`[gpt]`: result includes only the longest matching sequences (“g”), only the shortest ones (“p”), or all matching sequences (“t”);

`[fsr]`: the program inserts the output of the grammar in the text (“f”); the program replaces all matching sequences with the output of the grammar (“s”); the program ignores the output of the grammar (“r”);

`Limit`: the program stops after `Limit` number of matches. If `Limit` = 0 (zero), then there is no limit;

When applying the grammar to the text, the following vocabulary files may be used:

`S-Dic`: dictionary that contains all the simple words of the text;
`NAC-Dic`: dictionary that contains all the unambiguous compound words of the text;
`AC-Dic`: dictionary that contains all the (potentially ambiguous) compounds words of the text;

The result of the processing is the index of all matching sequences (plus eventually the corresponding output), stored in file `ResIndex`.

`recorind [cdl] [gpt] [fsr] Limit GRF Text S-Dictionary NAC-Dictionary AC-Dictionary Index Stats`

Applies the grammar GRF to the text file `Text`, using its index. The index of the text must be stored in the two special files “**idx**” (list of the tokens of the text) and “**ida**” (the positions of the occurrences of the tokens in the text); these two files must be located in the directory of the text, according to the INTEX application default. For instance, if the text is stored in the file:

```
c:\Intex\English\Corpus\foobar.snt
```

then the directory of the text is:

```
c:\Intex\English\Corpus\foobar_snt
```

and the two following index files must exist:

```
c:\Intex\English\Corpus\foobar_snt\idx
c:\Intex\English\Corpus\foobar_snt\ida
```

The grammar `GRF` is a Recursive Transition Network represented by a graph file (file “.grf” or “.cfg”), and may include references to embedded graphs. If the grammar has no output (or if the parameter “r” is used), the series of graphs is equivalent to a

Context-Free Grammar; if the graph contains no reference to embedded graphs, it is equivalent to a Finite-State Grammar.

`[cdl]`: type of the text file: “c”: concordance file (file format “.con”, usually produced by the `concord` program); “d”: text in which text units are delimited by the tag “{S}” (file format “.snt”, usually produced by the program `fst2txt`); “l”: raw Windows ANSI text (file format “.txt”, processed line by line);

`[gpt]`: result includes only the longest matching sequences (“g”), only the shortest ones (“p”), or all matching sequences (“t”);

`[fsr]`: the program inserts the output of the grammar in the text (“f”); the program replaces all matching sequences with the output of the grammar (“s”); the program ignores the output of the grammar (“r”);

`Limit`: the program stops after `Limit` number of matches. If `Limit` = 0 (zero), then there is no limit;

When applying the grammar to the text, the following vocabulary files may be used:

`S-Dic`: dictionary that contains all the simple words of the text;
`NAC-Dic`: dictionary that contains all the unambiguous compound words of the text;
`AC-Dic`: dictionary that contains all the (potentially ambiguous) compounds words of the text;

The result of the processing is the index of all matching sequences (plus eventually the corresponding output), stored in file `ResIndex`.

`table2fst MetaGraph Table Result`

Compiles a finite-state transducer or a context-free grammar (a graph stored in an “.fst” or an “.cfg” file) from a lexicon-grammar table `Table` and a meta-graph `MetaGraph`.

If the meta-graph and the lexicon-grammar include no reference to embedded graphs (such as “:NP”, “:Modifier” or “:AdvInsertion”), then the resulting transducer is a real finite-state transducer, and may be used by any of the `delae`, `recon`, `reconind`, and `fst2txt` programs.

If the meta-graph, or the lexicon-grammar table contain references to embedded graphs, then these references are left in the resulting “.cfg” file, and must be resolved during the application of the resulting transducer (by program `dicoe`, `recor`, `recorind`).

`tokenslist Index ResultingList`

When the INTEX application loads and index a text file (**Text > Open**), the list of the 100 most frequent tokens is displayed in the window "**Tokens list**". We saw earlier that INTEX processes four types of tokens:

-- *simple word forms* : sequences of letters between two delimiters;
-- *tags* : lexical entries written between the two special characters "{" and "}";
-- *digits* : the ten characters from "0" to "9";
-- *delimiters* : any other character but blanks (space, tab, newline and carriage return).

The program `tokenslist` takes the index of the text `Index`, and builds the list of all the tokens of the text `ResultingList`, in which each token is associated with its frequency, and stored alphabetically. As seen earlier, the text index is constructed by the program `indexer`; when using the INTEX graphical interface it is named "`idx`" and is stored in the text directory.

`tri [cdltr] Text ResText`

Sort program. This program takes the Alphabet file of the current language into account (it accesses environment variables INTEXLNG0, and then INTEXLNG, to locate the file). The ordering is described in the **Alphabet** file.

`[cdlt]`: type of the input text. "c": Delacf-type dictionary; "d": Delaf-type dictionary; "l" or "t": a line-delimited text. With option "l", delimiters are taken into account; with option "t", only letters count for the alphabetical ordering. Usually, "l" is used to sort lists of simple and compound words, whereas "t" is used to sort texts.

The resulting file `ResText` do not contain any duplicate.

`tri r column-number concordance result`

This variant of the sorting program is used specifically to sort concordance files (first parameter is "r").

`column-number`: this parameter has six possible values: 123, 132, 213, 231, 312, 321. These values correspond to the priority order of the columns during the comparisons. For instance, 132 means that lines will be sorted according to their first column (left context); if two lines are equal, then they will be sorted according to their third column (right context); if they have the same third column, then they will be sorted according to their second column (the matching sequence).

Note that the first column is always sorted from right to left.

`verifg [Ddl] mode FST SDic CDic NACDic Text RIndex Stats`

This program is an hybrid between the recognition program `recon` and the tagging program `etiqg`. It is used generally for debugging purposes, to check the application of a disambiguation grammar `FST` to a text `Text`, and look for inconsistencies.

If `mode = 1`, (corresponds to the option: "**display all matching sequences**" of the disambiguation module "Text > Disamb" of INTEX), the program runs exactly like recon: all the matching sequences are indexed.

If `mode = 0` (INTEX option "**Display inconsistencies between LGs and Text**"), then only sequences of the text that match with the input part of the disambiguation rule, but do not match with the output part of the grammar, are indexed.

The result of the processing is stored in the index `RIndex` ; this file has the same format as the result of the program `recon`.

The other parameters are identical to the ones of the program `etiqg`:

The first parameter corresponds to the type of the input text:

-- "D": the input is a dictionary, in which each entry is a text that needs to be tagged (e.g. a DELAC-type dictionary contains compound word sequences in the entry field; these entries need to be tagged in order to produce the corresponding DELACF-type dictionary);
-- "d": the input is a text in the INTEX format: "`.snt`";
-- "l": the input is a text that is line delimited (NEWLINE or CARRIAGE-RETURN, NEWLINE).

The third parameter `FST` is the local grammar to check against the text. When using the graphical interface of INTEX, this FST is usually the union of all the local grammars that are selected in the **Disamb** panel. The two next parameters are the vocabulary files of the text: `SDic` includes the simple words and `CDic` includes the compounds (they are usually created by `dicos` and `dicoc`). The next parameter `NACDic` is a dictionary that contains unambiguous compounds. Some information (number of disambiguated sequences, number of inconsistencies between the vocabulary files and the disambiguation grammar) is stored in the resulting file `Stats`.

Chapter 18. FILE FORMATS

Alphabet

The file is read by the quasi-totality of programs, in a transparent fashion, beginning with the environmental variables INTEXLNG0 and INTEXLNG (which contain the names of current language files).

The alphabet file is a Windows ANSI type file (ASCII extended to 8 bits), in which each letter is inventoried and described.

The two first lines of the file are elective: they give the name and the font sizes of the characters used to display texts (in general, a font size of proportional width), concordances and dictionaries (in general, of a fixed font size).

If these two lines are not present, INTEX will use the default font size, used by Windows to display text (we can modify this font in the display section of the control panel). If only one line is present, INTEX will use the same font size to display text, dictionaries and concordances.

These two lines must imperatively begin with the special character "#" (pound). For example:

```
#"Times New Roman Greek" 11
#"Courier Greek" 11
```

After which, for each letter, there are two possibilities:

-- if the letter has no accent, we write the upper case, followed by the lower case letter, for example. :

Aa

-- if the letter takes an accent, we write the upper case letter without the accent, followed by the upper case letter with accent, followed by the lower case letter, for example. :

AÀà

This double representation is necessary in French to describe the fact that the letter "A" can represent the letter "à" in numerous texts (the upper case letters are not always accented). Two cases in particular:

-- if the upper case letters are **always** accented in the texts in question, then it is sufficient to enter each letter, accented or not, with it’s accented form, for example. :

Aa
Àà
Ââ
Ää
Bb
...

-- if the upper case letters are **never** accented in the texts in question, then it is sufficient to enter each letter, accented or not, with it’s non-accented form, for example. :

Aa
Aà
Aâ
Aä
Bb
...

In these two cases, the alphabet contains no rows of three columns. The order in which letters are inventoried in the **Alphabet** file is used by all of INTEX’s sorting programs. Since a capitalized form in the text can correspond to a lower case lexical entry, all programs that consult the dictionaries use the relation between capital and lower case letters.

Some constraints:

-- all letters must be represented by an byte: INTEX cannot process UNICODE codes (where each letter is represented by two bytes), or codes with variable length in which certain letters have a longer code than others: for example. "{e\acute}" to represent the letter "é"

-- the special ASCII characters, like "0" (which represents the end of the character chain in C) or "26" which represents the end of the file in Windows cannot represent letters.

Texts ".txt" and ".snt" format

Texts in ".txt" Windows ANSI format, are texts in which all letters are represented by an byte (ASCII extended to 8 bits). Let’s recall certain format constraints:

-- INTEX cannot process UNICODE files; all the letters must be represented by an;

-- Special ASCII characters like "0" which represents the end of the character chain in C) or "26" which represents the end of the file in Windows cannot appear in a ".txt" file;

-- the "New line" character, or the sequence of characters "Carriage return, New line" are, by default, interpreted by INTEX grammars as representing paragraph delimiters (rather than line delimiters). One must be attentive to the coherence between their usage in the texts and their interpretation by INTEX.

Texts in ".snt" format, are texts which can contain INTEX tags. INTEX tags are sequences of characters between two brackets. Most INTEX programs use the "{S}" tag to represent the separator between two textual units (generally sentences). For this reason, a file in ".snt" format must absolutely not contain the characters "{" and "}".

The other format constraints are described in the chapter on format and standardizing the texts.

Concordances: files ".con"

INTEX concordances are represented by Windows ANSI files. Each line of the concordance has four columns:

-- the left context,
-- a tabulation character,
-- the indexed sequence,
-- a tabulation character,
-- the right context,
-- a tabulation character,
-- the address (position) of the sequence in the text,
-- a space,
-- the length of the sequence in the text.

In general, the address and the length of the sequences are not displayed. They are used to link the concordance line to the text (for example, if the user clicks on the line in the concordance, INTEX will display the corresponding occurrence in the text).

A concordance cannot contain an empty line.

Dictionaries in ".dic" format: DELAS, DELAF, DELACF

The dictionaries, in ".dic" format, are Windows ANSI files, in which each lexical entry is represented on one line. A dictionary cannot contain an empty line. Here is a typical entry:

cousines germaines,cousin germain.N+NA+Hum:fp/version 1.0 du DELACF

More generally, each line has the following format:

-- the dictionary entry, ex. *cousines germaines*
-- a comma,
-- a canonical form associated with the entry, ex. *cousin germain*
-- a period,
-- a category code in upper case, ex. *N*
-- syntactic or semantic codes prefixed with the character "+", ex. *+Hum+Politique*
-- one or more series of inflectional prefixes with the character ":", ex. *:P3s*
-- a zone for comments introduced by the character "/", ex. */found in the GDEL*

If the canonical form is identical to the dictionary entry, it is not necessary to mention it; the syntactic codes, the inflectional codes, and the comments are optional.

All of the dictionary entries must be sorted in an alphabetical order which is perfectly compatible with that described in the alphabet of the current language. We can use the `sort` program or the INTEX **DELA > Sort** command to ensure that a dictionary is correctly sorted.

The format of DELA dictionaries is described in the chapter on electronic dictionaries. The grammars that we find in the folder Dic-utils of the current language can be used by the `recondic` program, the INTEX **DELA > Check Format** command can be used to verify the format of a dictionary.

Compressed dictionaries: ".bin" and ".inf" files

".dic" type dictionaries can be compressed thanks to the `dic2fst` program or the INTEX **DELA > Compress into FST** command. The result of the compression is stored in two files: ".bin" represents the automaton of the dictionary, and ".inf" houses the lexical information of the dictionary. The ".bin" file is a "binary" file which cannot be edited; the file ".inf" is a Windows ANSI file, that we can edit with a word processing or text-editing program.

".bin" file:

-- four bytes that represent the size of the automaton by number of bytes;

-- then, for each state of the automaton:
-- if it is not a terminal state:
-- 1 bit: 1 = not terminal;
-- 7 bits: number of outgoing transitions;
-- if it is a terminal state:
-- 1 bit: 0 = is terminal;
-- 7 bits: number of outgoing transitions;
-- index of information associated with the lexical entry coded on 3 bytes
-- then for each outgoing transition
-- an byte for the transition tag (the character)
-- three bytes for the address of the destination state.

".inf" file:

-- the amount of information is different i.e. the size of the alphabet coming out of the transducer (written in ASCII, followed by a line change);

-- each bit of information is written on a single line; they are implicitly numbered beginning at 1; the number corresponds to the index produced by the terminal states of the automaton.

To be able to factor a maximum number of lexical information items, only the difference between lexical entries and canonical forms is represented. For example, the two following dictionary entries:

```
aidons,aider.V+t:P1p
aimons,aimer.V+t:P1p
```

will be associated to the following notation of lexical information:

```
3er.V+t:P1p
```

"3er" signifies: erase the three last letters; add "er". in similar fashion, for the following compound lexical entries:

```
cousines germaines,cousin germain.N+Hum:fp
déléguées syndicales,délégué syndical.N+Hum:fp
```

the lexical information would be the same for the two entries:

```
2 2.N+Hum:fp
```

"2 2" signifies: erase the last two letters of the two constituents. This technique allows us to considerable reduce the size of the alphabet that comes out of the dictionary transducer, and allows an important gain in terms of minimizing the automaton.

".fst" transducer

"`.fst`" files are built by the programs `re2fst` (based on a rational expression) and `gr2fst` (based on a graph, or a series of graphs).

These files are in Windows ANSI format (we can edit them); they contain the following information:

-- the number of states:
-- the size of the alphabet;
-- the alphabet itself; the letters of the alphabet are either in chains of characters used for identification, or pairs of character chains a/b, where a is used for the identification and b is used for the production; the letters are separated by the character "%"; they are implicitly numbered beginning at 0;
-- then, for each state of the transducer (the states are implicitly numbered beginning with 1):
    -- the first character is "t" if the state is terminal, ":" if not;
    -- each transition is given in the form of two wholes: the number of the alphabet letter, and the number of the destination state in the transition;
    -- the last transition is followed by the relative whole -1;
    -- the last state is followed by the character "f".

For example, the following transducer:

```
3 5
%a%b/z%c%
: 0 2
: 1 3 0 2
t 2 3
f
```

represents a transducer which has 3 letters and 5 states; the three letters are "`a`" (numbered 0), "`b/z`" (numbered 1) and "`c`" (numbered 2). A transition begins at state 1, is tagged by "a" and proceeds to state 2; two transitions emerge from state 2; the first is tagged b/z and proceeds to state 3; the second is tagged "a" and proceeds to state 2; State 3 is terminal; a transition leaves from state 3; it is tagged "c" and proceeds to state 3.

".mft" Multiple transducer files

".mft" files represent text in the form of transducers; one transducer per phrase. The transducers are simply written one following the others. (exactly as in an ".fst" file), and are separated by a header line which gives the number of each phrase.

".grf" graphs

".grf" files represent INTEX graphs. These files are in Windows ANSI format can be edited. Each graph file includes a header beginning with the line:

```
#FSGraph 4.3
```

(the version number is not necessarily 4.3), and ending with a line containing nothing but the character "#". In the header, the graph's parameters of belonging are indicated: Size of the graph (SIZE), name and size of the fonts in the nodes (FONT), and beneath the nodes (OFONT), background color (BCOLOR), font color (FCOLOR), auxiliary node color (ACOLOR), selection color (SCOLOR) and comment node colors (CCOLOR). The following options:

DBOX	displays boxes surrounding the transition tags
DFRAME	displays a frame
DDATE	displays the graph's last save date
DFILE	displays the graph's file name
DDIR	displays the graph's directory (folder) name
DRIG	text in nodes are written from right to left
DRST	nodes tagged with "<E>" are represented by small circles (states)
FITS	scale of the graph (100 = 100 %)
PORIENT	the graph is displayed in Portrait "P" or Landscape "L".

Next, the number of nodes in the graph is indicated; the nodes are implicitly numbered beginning at 0. Node 0 is the initial node of the graph; node 1 is the terminal node. For each node, we give it's tag, it's coordinates in the map, the number of incidental connections (0 if there are no connections), then, for each connection, the number of incidental states..

Here is an example of a ".grf" file:

```
#FSGraph 4.0
SIZE 1672 1292
FONT Times New Roman:  12
OFONT Times New Roman:B 12
BCOLOR 16777215
FCOLOR 0
ACOLOR 12632256
SCOLOR 16711680
CCOLOR 255
DBOXES y
DFRAME n
DDATE y
DFILE y
DDIR n
DRIG n
DRST n
FITS 100
PORIENT L
#
3
```

```
"A/B" 120 36 1 2
"" 272 36 0
"C" 212 36 1 1
```

This file represents the following graph:

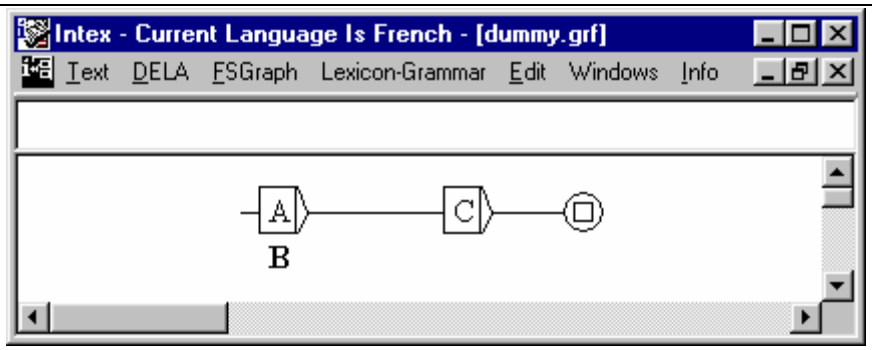


Figure 112. Representation of a graph

Index of the text: "idx" and "ida" files

idx and ida files are constructed by the `indexer` program. These files represent the index of all lexemes in a text. Let's remember that the lexemes are the basic objects beginning with which, INTEX programs can recognize the linguistic units. They are:

-- adjoining sequences of letters (simple forms) ;
-- tags written between brackets; they represent linguistic information; these tags cannot be present in an ".snt" format text (otherwise the brackets are not treated as special characters) ;
-- digits (characters between "0" and "9") ;
-- separators, i.e. all characters that are neither letters nor digits.

Characters: space (SPACE), line change (NEW LINE and CARRIAGE RETURN characters) and the tabulation character (TAB) are not indexed. The code with the value 0 (which represents the end of a chain of characters) cannot be indexed.

The idx file

This file contains all the lexemes of the text, sorted in alphabetical order. For each lexeme of the index (key), the file contains:

-- the sequence of characters of the lexeme (an byte for the digits and separators, several bytes for the simple forms, and the tags) ;

-- the 0 value code ('\0' in C language),

-- the number of necessary bytes *nbo* to represent the relative addresses of all occurrences of the key in the text, itself coded on an byte. The address of an occurrence of a key in the text is, in effect, represented by the difference between this

address and the preceding occurrence. The number of necessary bytes *nbo* depends on the maximum distance *dmax* between two consecutive occurrences of the lexeme:

if *dmax* < 256 characters, then *nbo* = 1 ;
if *dmax* < 256 x 256 = 65536 characters, *nbo* = 2 ;
if *dmax* < 256 x 256 x 256 = 16 777 216, *nbo* = 3 ;
otherwise, *nbo* = 4.

-- the number of occurrences of the key in the text, coded in three bytes;

-- the address of the first occurrence of the key in the text, coded on bytes.

The `ida` file

This file contains the list of all the addresses of each key in the text, expressed in relative. The number of bytes necessary to represent the addresses of each key is given in the `idx` file.

Index of recognized sequences "ind"

This file is constructed by the programs `dicoc`, `dicoe`, `recon`, `reconind`, `recor`, `recorind` and `verifg`. It contains the index of all recognized sequences.

The index is a file in Windows ANSI format (editable).

-- the first character represents the type of operation to be performed on the text, i.e. how those items produced by the transducer will be used.: "f" (fusion) signifies that the sequences produced by the transducer are inserted into the text; "s" (substitution) signifies that the sequences recognized by the transducer are replaced by the produced sequences; "r" (do nothing) signifies that we can disregard the sequences produced by the transducer. It's this last option that is generally used when applying a finite automaton (a non-producing transducer).

Next, for each sequence recognized (indexed), we give:

-- the address of the sequence in the text, relative to the preceding indexed sequence; it amounts to the difference between the position of the sequence and that of the preceding indexed sequence. Note that this whole can be null in the case where a sequence is identified several times with different productions, or if several identified sequences begin at the same position in the text;

-- the length of the identified sequence (by number of characters); this whole cannot be null ;

-- the sequence produced by the transducer (this sequence can be empty); the sequence is a chain of characters ending with the code 0 ('\0' in C language).

An example of an `ind` file:

```
r
250 2 il
123 8 la table
0 16 la table de Paul
```

This file represents three sequences, indexed using the automaton ("r" signifies that we disregard the eventual produced texts); at position 250, "il" was indexed; at position 250 + 123 = 373, the two sequences "la table" and "la table de Paul" were indexed.

Chapter 19. REFERENCES

We cannot provide an exhaustive bibliography that would list the numerous projects that are either related to the construction of INTEX modules (Bulgarian, English, French, Greek, Korean, Italian, Portuguese, Spanish, Serbo-Croatian, Thai), or that are based on the use of INTEX (as a tool to process corpora or to perform information extraction, or as a development environment to build NLP computer applications).

Although the following bibliography mentions also studies that are not directly related to INTEX, we feel that it is a very useful source for linguists and INTEX users:

Leclère, Christian. 1998. "Travaux récents en Lexique-grammaire". In "Le Lexique-grammaire", Béatrice Lamiroy (ed.), Travaux de Linguistique n° 37, Louvain-la-Neuve : Duculot, pp. 155-186.

19.1. Main References (Books and theses)

The following book presents the French DELA database:

Courtois Blandine, Silberztein Max Eds, 1990, *Les dictionnaires électroniques du français*. Langue Française #87. Larousse: Paris (127 p.).

The following books show how Finite-State technology can successfully used to represent linguistic phenomena and to build NLP applications.

Gross Maurice, Perrin Dominique. 1989. *Electronic Dictionaries and Automata in Computational Linguistics, Lecture Notes in Computer Science #377.* Springer: Berlin/New York.

Roche Emmanuel, Schabes Yves Eds. 1997. *Finite-State Language Processing*, Cambridge, Mass./London, The MIT Press.

The following thesis presents the first attempt to compress a full DELAF dictionary into a minimal, deterministic finite-state automaton:

Revuz Dominique. 1992. Minimization of acyclic deterministic automata in linear time. *Theoretical Comput. Sci.*, vol. 92, n# 27 1, pp. 181-189.

The following book presents the DELA system of dictionaries and the INTEX programs:

Silberztein Max. 1993. *Dictionnaires électroniques et analyse automatique de textes. Le système INTEX*. Masson: Paris (240 p.).

19.2. Articles

Chrobot Agata, Courtois Blandine, Hammani Mary, Gross Maurice, Zellagui Katia. 1999. *Dictionnaire Electronique DELAC anglais : noms composés*. Technical report #59. LADL, Université Paris 7: Paris.

Courtois Blandine, Silberztein Max. 1989. Les dictionnaires électroniques DELAS et DELAC. In *RELAI: Recherches en Linguistique Appliquée à l'Informatique.* Université Laval: Québec.

Courtois Blandine. 1990, Un système de dictionnaires électroniques pour les mots simples du français, in *Les dictionnaires électroniques du français*. Langue Française #87. Larousse: Paris.

Courtois Blandine, Garrigues Mylène, Gross Gaston, Gross Maurice, Jung René, Mathieu-Colas Michel, Silberztein Max, Vivès Robert. 1997. *Dictionnaire électronique des noms composés DELAC : les composants NA et NN,* Rapport Technique du LADL 55, Paris, Université Paris 7.

Courtois Blandine, Garrigues Mylène, Gross Gaston, Gross Maurice, Jung René, Mathieu-Colas Michel, Monceaux Anne, Poncet-Montange Anne, Silberztein Max, Vivès Robert. 1997. *Dictionnaire électronique DELAC : les noms composés binaires,* Rapport Technique du LADL 56, Paris, Université Paris 7.

Friburger Nathalie, Silberztein Max 1999. Le débogueur de grammaires sous INTEX. In *Analyse lexicale et syntaxique : le système INTEX*, Cédrick Fairon Ed. Linguvisticae Investigationes vol. XXII, pp. 413-423 : 1998-1999.

Gross Gaston. 1988. *Noms composés N de N.* Rapport de Recherches 5, Laboratoire de Linguistique Informatique, Villetaneuse : Université Paris 13.

Gross Gaston. 1988. *Noms composés N de N.* Rapport de Recherches 6, Laboratoire de Linguistique Informatique, Villetaneuse : Université Paris 13.

Gross Gaston. 1988. Degré de figement dans les noms composés. *Langages* 90, pp. 57-72, Paris: Larousse.

Gross Gaston. 1990. Définition des noms composés dans un lexique-grammaire. *Langue Française* 87, Paris : Larousse.

Gross Maurice. 1986. "Lexicon-Grammar. The Representation of Compound Words". In *COLING-1986. Proceedings*, Bonn, pp. 1-6.

Gross Maurice. 1986. *Grammaire transformationnelle du français. 3 - Syntaxe de l'adverbe*, Paris, 670 p.

Gross Maurice. 1989. The Use of Finite Automata in the Lexical Representation of Natural Language. In *Electronic Dictionaries and Automata in Computational Linguistics, Lecture Notes in Computer Science 377*, pp. 34-50, Berlin/New York: Springer.

Gross Maurice. 1993. Les phrases figées en français. In *L'information grammaticale*, pp. 36-41, Paris.

Gross Maurice. 1997. The Construction of Local Grammars, in E.Roche et Y.Schabes (eds.), *Finite-State Language Processing*, Cambridge, Mass./London, The MIT Press, pp. 329-352.

Klarsfeld Gaby, Hammani Mary. *Dictionnaire électronique du LADL pour les mots simples de l'anglais*. DELAS v4. Technical report. LADL, Université Paris 7: Paris.

Mathieu-Colas Michel. 1987. *Composés de type NAdj.* Rapport de Recherches 3, Laboratoire de Linguistique et Informatique, Université de Villetaneuse.

Mathieu-Colas Michel. 1988. Variations graphiques des mots composés dans le Petit Larousse et le Petit Robert. *Lingvisticae Investigationes* XII:2, pp. 235-280, Amsterdam/Philadelphia : John Benjamins.

Meunier Annie. 1979. Some remarks on French colour adjectives. In *SMIL, Journal of Linguistic Calculus*, pp. 148-165, Stockholm: Skriptor.

Muller Claude, Royauté Jean, Silberztein Max Eds. 2004. *INTEX pour la Linguistique et le Traitement Automatique des Langues,* Presses Universitaires de Franche-Comté (400 p.).

Piot Mireille. 1988b. Conjonctions de subordination et figement. *Langages* 90, pp. 39-56, Paris: Larousse.

Roche Emmanuel. 1997. Parsing with finite state transducers. In E. Roche and Y. Schabes Eds. *Finite-State Language Processing*, Cambridge, Mass./London, The MIT Press, pp. 241-281.

Silberztein Max. 1989. The lexical analysis of French, in *Electronic Dictionaries and Automata in Computational Linguistics, Lectures Notes in Computer Science* #377, Berlin/New York: Springer.

Silberztein Max. 1990. Le dictionnaire électronique des mots composés. In *Dictionnaires électroniques du français*, Courtois Blandine, Silberztein Max Eds. *Langue Française* #87, pp. 71-83. Larousse: Paris.

Silberztein Max. 1991. A new approach to tagging: the use of a large-coverage electronic dictionary, *Applied Computer Translation* 1(4).

Silberztein Max. 1992. Finite state descriptions of various levels of linguistic phenomena, *Language Research* 28(4), Seoul National University, pp. 731-748.

Silberztein Max. 1994, Les groupes nominaux productifs et les noms composés lexicalisés, *Lingvisticae Investigationes* XVII:2, Amsterdam/Philadelphia : John Benjamins, p. 405-426.

Silberztein Max. 1994. INTEX: a corpus processing system, in *COLING 94 Proceedings*, Kyoto, Japan.

Silberztein Max. 1996. *Levée d'ambiguïtés avec INTEX*, in BULAG #21. Université de Franche Comté.

Silberztein Max. 1996. *Analyse automatique de corpus avec INTEX*, in LINX #34-35 : *Hommage à Jean Dubois*. Université Paris X: Nanterre.

Silberztein Max. 1997. The Lexical Analysis of Natural Languages, in *Finite-State Language Processing*, E. Roche and Y. Schabes (eds.), Cambridge, Mass./London, MIT Press, pp. 175-203.

Silberztein Max. 1999. *Transducteurs pour le traitement automatique des textes*, in *Travaux de Linguistique*. Béatrice Lamirot Ed. Duculot, 1999.

Silberztein Max. 1999. *INTEX: a Finite State Transducer toolbox*, in Theoretical Computer Science #231:1, pp. 33-46.

Silberztein Max. 1999. *Indexing large corpora with INTEX*, in Computer and the Humanities #33:3.

Silberztein Max. 1999. Les graphes INTEX. In *Analyse lexicale et syntaxique : le système INTEX*, Cédrick Fairon Ed. Linguisticae Investigationes vol. XXII, pp. 3-29.

Silberztein Max. 1999. Les expressions figées dans INTEX. In *Analyse lexicale et syntaxique : le système INTEX*, Cédrick Fairon Ed. Linguisticae Investigationes vol. XXII, pp. 425-449.

Silberztein Max. 2003. Finite-State Recognition of French Noun Phrases. In *Journal of French Language Studies*. vol. 13:02, pp. 221-246. Cambridge University Press.

19.3. Proceedings of the INTEX Workshops

Fairon Cédrick Ed., 1999. *Analyse lexicale et syntaxique: le système INTEX*, Actes des Premières et Secondes Journées INTEX. Linguvisticae Investigationes vol. XXII.

Dister Anne Ed., 2000. Actes des Troisièmes Journées INTEX. Liège 2001. In *Informatique et Statistique dans les Sciences Humaines*. Université de Liège, n° 36.

Royauté Jean, Silberztein Max Eds. 2004. *INTEX pour la Linguistique et le Traitement Automatique des Langues*. Proceedings of the 4th and 5th INTEX workshop, Bordeaux, May 2001, Marseille, May 2002: Presses Universitaires de Franche-Comté (400 p).