NooJ Manual

(revised 2020/06/09)

Please cite this document as:

Silberztein Max, 2003-. *NooJ Manual*. Available for download at: www.nooj-association.org

Please report any error or request to: max.silberztein@univ-fcomte.fr

This volume explains *how* to use the NooJ software. To understand its theoretical and methodological background and learn more about *why* it is like it is, see:

Silberztein Max, 2016. *Formalizing Natural Languages: the NooJ approach*. Wiley Eds.

Table Of Content

GETTING STARTED

This first section presents NooJ and its applications, takes you through the installation process, and then gives you the minimum amount of information necessary to launch a basic search in a text.

1. Welcome

Introduction

NooJ is a development environment used to construct large-coverage formalized descriptions of natural languages, and apply them to large corpora, in real time. The descriptions of natural languages are formalized as electronic dictionaries, as grammars represented by organized sets of graphs.

NooJ supplies tools to describe inflectional and derivational morphology, terminological and spelling variations, vocabulary (simple words, multi-word units and frozen expressions), semi-frozen phenomena (local grammars), syntax (grammars for phrases and full sentences) and semantics (named entity recognition, transformational analysis). In effet, NooJ allows linguists to combine in one unified framework Finite-State descriptions such as in XFST, Context-Free grammars such as in GPSG, Context-Sensitive grammars such as in LFG and unrestricted grammars such as the ones developped in HPSG.

NooJ is also used as a corpus processing system: it allows users to process sets of (thousands of) text files. Typical operations include indexing morpho-syntactic patterns, frozen or semi-frozen expressions (e.g. technical expressions), lemmatized concordances and performing various statistical studies of the results.

System requirements

There are two versions of NooJ: a .NET application that runs natively on any Windows PC, and a JAVA application that runs on any computer via the JAVA virtual machine. The JAVA application is GPL open source and is distributed by the European Metashare platform at:

http://metashare.nlp.ipipan.waw.pl/metashare/repository/browse/source-code-for-the-java-version-of-nooj/2f8caa506aff11e2aedc000423bfd61c0a125e4434514b43ba542943a6108ec7/

-- The Java version is a simplified version of NooJ; it contains all NooJ major functionalities, and its available source code is simplified (more readable). It is easier to use for demos and for pedagogical applications. If you have 20 students in a computer lab room, some of them with a LINUX PC, others with a Macbook, etc. just use the Java version.

-- The .NET version is more powerful (more complicated to use) and has some helpful functionalities for more ambitiuous projects. For instance, the .NET version can process a large number of text files at the same time, whereas the JAVA version only processes one text file at a time. The .NET version is also more optimized (faster). If you want to build a real-world NLP application, or if you want to develop large-coverage linguistic resources, this is the version you should use.

If you are planning to use NooJ to develop complex grammars in a graphical form, a good screen is a must: at least a 19-inch screen, with HD resolution, and a minimum of 60 Hz refresh rate.

NooJ after INTEX

INTEX's technology was based on my thesis work (1989), and its first version was released in 1992. Between 1992 and 2002, INTEX has substantially evolved, mostly "organically", in response to the needs of its users. Moreover, along the years, INTEX's technology has become obsolete: written in C/C++, it was monolingual, could only handle one single file format, one text file at a time, no support for XML, etc.

In 2002, while I was abroad, former colleagues who had total access to INTEX's sources and linguistic resources "developed" the Unitex software. In fact, Unitex is just an unauthorized copy of INTEX's sources, of its methodology, algorithms and its GUI, see:

http://www.nooj-association.org/index.php?option=com_content&view=article&id=83&Itemid=650

Without any possibility to defend my 10-year work (which belongs to my employer), I decided then to develop a new, much more ambitious and better INTEX: NooJ.

NooJ has been rewritten from the ground up, using .NET Object Oriented architecture. However,

-- I have used my 10-year experience as the INTEX's author to redesign the whole system, making sure to redesign every single "good" feature, while avoiding to make the same mistakes, e.g. when we had to hack INTEX to fit in a new, important functionality that should not be a hack in the first place...

-- I have made sure that NooJ is a much more natural, easier linguistic environment than INTEX. Note that in general, linguists' work is much easier with NooJ than with INTEX and the differences between INTEX and NooJ are always in the direction of simplification: for instance, NooJ has no more DELAF/DELACF dictionaries, NooJ handles multiwords' inflection in a very simple way, NooJ's grammars are self-contained (instead of being made of dozen of different files), morphological and syntactic grammars are sharing the same linguistic engine, NooJ offers "free" transformation capability (i.e. NooJ can automatically produce paraphrases, without the need for linguistis to write transformational grammars), etc.

To program NooJ, I decided to follow a Component-Based Software approach, which is a step beyond the Object-Oriented Programming paradigm. The .NET framework gave NooJ a number of great functionalities, including the automatic management of memory, support for hundreds of text encodings and formats (including Microsoft Office), native XML compatibility, both for parsing XML documents and to store objects (XML/SOAP), etc.

-- NooJ has a non-destructive linguistic engine: NooJ, as opposed to INTEX, never modifies the texts it is processing (no more REPLACE nor MERGE modes). Therefore, NooJ is an ideal tool to perform a large number of operations in cascade or in parallel. Grammar writers need no longer enter meta-data in their grammar to process the results of previous analyses, and NooJ grammars, as opposed to INTEX's, are small and truly independent from each other (INTEX's grammars had the tendency to become huge very quickly because there was no simple way to adapt them to each particular need).

-- NooJ's dictionaries are a great enhancement over INTEX DELA-type dictionaries as well as lexicon-grammar tables. A NooJ dictionary is similar to DELAS-DELAC dictionaries (no more difference between simple and multiword units) and can represent spelling and terminological variants as well (no more need for a separate DELAV dictionary). NooJ does not need DELAF-DELACF-type "inflected dictionaries" because it processes word inflection transparently, including multiword units' inflection (INTEX did not process compound word inflection). Moreover, NooJ generalizes inflectional morphology to derivational morphology, so that derivations are formalized in a very natural way (there was no provision for derivational morphology in INTEX).

-- NooJ's integration of morphology and syntax allows NooJ to perform morphological operations inside syntactic grammars: for instance, it is possible to ask

NooJ to locate all verbs conjugated in the present, third person singular, and replace them with the wordform "is" followed by the same verbs in their Past Participle form. In the sme manner, we can transform the sentence "the students demonstrate" with "the student's demonstration" simply by nominalizing the verb (which is performed by a simple operator such as $V_N+s). Now we can write and perform automatic transformations on large texts!

-- NooJ processes lexicon-grammar tables without meta-graphs (INTEX's meta-graphs were very cumbersome to use, could not be merged with phrasal syntactic grammars, and more importantly, could not be merged together because the explosion of the size of their compiled form). Meta-Grammars are the pleague of modern Computational Linguistics: most researchers who use limited formal frameworks (such as XFST or TAG) are forced to use "meta-grammars" and "meta compilers" that produce thousands of grammars automatically to overcome the limitations of their tool: no such thing in NooJ.

-- etc.

In conclusion, NooJ is a radically better lingusitic platform than INTEX/Unitex. I hope that the new engine and functionalities (not even mentionning Ethics) will give NLP developers excellent reasons to try NooJ!

Programming NLP applications with NooJ

One can easily build prototypes that contain powerful NooJ functionalities.

NooJ's functions are available via a command-line program: **`noojapply.exe`**, which is installed in the NooJApp.zip folder. The command **`noojapply.exe`** can be called either directly from a "SHELL" script, or from more sophisticated programs written in PERL, C++, JAVA, via a system () command. Note that **`noojapply.exe`** can be used on any UNIX or LINUX PC via either their .NET or MONO virtual machines.

`noojapply.exe` allows users to apply dictionaries and grammars to texts automatically. The result is an index that lists all the matching sequences together with their corresponding output.

Note INTEX users: **`noojapply.exe`** provides the same functionalities as the 30+ programs that constituted the INTEX package, in a much more efficient way. For instance, noojapply processes any number of texts (instead of a single one) and it compiles deterministic grammars dynamically, etc.

Text Annotation Structure

NooJ's linguistic engine uses an annotation system. An annotation is a pair (*position*, *information*) that states that a certain sequence of the text has certain properties. When NooJ processes a text, it produces a set of annotations, stored in the **Text Annotation Structure** (TAS); annotations are always kept synchronized with the original text file, which is itself never modified. Annotations can be associated to single wordforms (e.g. to annotate "table" as a noun), to parts of wordforms (e.g. to annotate "not" in "cannot" as an adverb), to multi-word units (e.g. to annotate "round table" as a noun) as well as discontinuous expressions (e.g. to annotate "take into account" in "John took the problem into account"). Annotations store information that can represent anything: lexical information, morphological information, syntactic information, semantic information, pragmatic information, etc.

NooJ morphological and syntactic parsers provide tools to automatically add annotations to a TAS, to remove (filter out) annotations from a TAS, to export annotated texts and corpora as XML documents, as well as to parse XML documents and import certain of their XML tags into NooJ's own TAS.

Computational devices

NooJ's linguistic engine includes several computational devices used both to *formalize* linguistic phenomena and to *parse* texts.

Finite-State Transducers

A finite-state transducer (FST) is a graph that represents a set of text sequences and then associates each recognized sequence with some analysis result. The text sequences are described in the **input** part of the FST; the corresponding results are described in the **output** part of the FST.

Typically, a syntactic FST represents word sequences, and then produces linguistic information (such as its phrasal structure). A morphological FST represents sequences of letters that spell a wordform, and then produces lexical information (such as a part of speech, a set of morphological, syntactic and semantic codes).

NooJ contains a graphical editor (as well as a dozen tools) to facilitate the construction of FSTs and their variants (FSA, RTNs and ERTNs).

Finite-State Automata (FSA)

In NooJ, **Finite-State Automata** are a special case of finite-state transducers that do not produce any result (i.e. they have no output). NooJ's users typically use FSA to locate morpho-syntactic patterns in corpora, and extract the matching sequences to build indices, concordances, etc.

Recursive Transition Networks (RTNs)

Recursive Transition Networks are grammars that contain more than one graph; graphs can be FST or FSA, and also include references to other, embedded graphs; these latter graphs may in turn contain other references, to the same, or to other graphs. Generally, RTNs are used in NooJ to build libraries of graphs from the bottom-up: simple graphs are designed; then, they are re-used in more general graphs; these ones in turn are re-used, etc.

RTNs provide a simple and elegant way to develop large and complex Context-Free Grammars.

Enhanced Recursive Transition Networks (ERTNs)

Enhanced Recursive Transition Networks are RTNs that contain variables and constraints; these variables typically store parts of the matching sequences, and then are used to perform some operation with them (e.g. compute their plural form or their translation in another language), or to satisfy some constraints (e.g. make sure a verb agrees with its subject).

Variables and constraints give NooJ the power of an Linear-Bounded Automaton (LBA), which in effect gives linguists the ability to construct Context-Sensitive Grammars. NooJ's grammars have the same expressive power as LFG grammars.

Because variables' content can be duplicated, inserted, deleted or displaced, applying ERTNs allows NooJ to perform linguistic **transformations** on texts: we will see that although this gives the NooJ the power of a Turing Machine, we have found another, more elegant way to perform transformations.

Examples of transformations include negation, passivization, nominalization, etc. Graphical FSA, FSTs, RTNs and ERTNs have their "written" counterparts: formal grammars in the form of formation rules: Regular expressions, Context-Free Grammars and Context-Sensitive Grammars.

Regular Expressions

Regular Expressions constitute also a quick way to enter simple queries without having to draw a FSA. When the sequence to be located consists of a few words, it is much quicker to enter these words directly into a regular expression. However, as the query becomes more and more complex as is usually the case in Linguistics, one should build a grammar.

Context-Free Grammars (CFGs)

In NooJ, CFGs constitute an alternative means to enter morphological or syntactic grammars that would be drawn via an RTN.

For instance, NooJ includes an inflectional/derivational module that is associated with its dictionaries, so that it can automatically link dictionary entries with their corresponding forms that occur in corpora (there is no DELAF/DELACF dictionary in NooJ).

NooJ dictionaries generally associate each lexical entry with an inflectional and/or derivational paradigm. For instance, all the verbs that conjugate like "aimer" are linked to the paradigm "+FLX=AIMER"; all the verbs that accept the "-able" suffix are linked to the paradigm "+DRV=ABLE", etc.

Paradigms such as "AIMER" or "ABLE" are described either graphically in RTNs or by CFGs in text files.

Context-Sensitive Grammars (CSGs)

In NooJ, CFGs constitute an alternative means to enter morphological or syntactic grammars that would be drawn via an ERTN. Basically CSGs are CFGs in which we use variables (to store pieces of the text input) as well as constraints (to check agreements between variables).

Linguistic Resources

With NooJ, linguists build, test and maintain two basic types of linguistic resources:

-- **Dictionaries** (**`.dic`** files) usually associate words or expressions with a set of information, such as a category (e.g. "Verb"), one or more inflectional and/or derivational paradigms (e.g. how to conjugate verbs, how to nominalize them), one or more syntactic properties (e.g. "+transitive" or +N0VN1PREPN2), one or more semantic properties (e.g. distributional classes such as "+Human", domain classes such as "+Politics"). Lexical Properties can be binary, such as "+plural" or can be expressed as an attribute-value pair, such as "+gender=plural". Values can belong to the meta-language, such as in "+gender=plural", to the input language such as in "+synonym=pencil" or to another language, such as in "+FR=crayon".

NooJ's dictionaries constitute a converged and enhanced version of the DELA-type dictionaries and lexicon-grammars that were used in INTEX: a NooJ dictionary can include simple words (like a DELAS), multi-word units (like a DELAC) and can link lexical entries to a canonical form (like a DELAV). NooJ's dictionaries can store syntactic and semantic information (just like a lexicon-grammar) as well as multilingual information and thus be used in a MT system. Contrary to INTEX, NooJ does not need full inflected form dictionaries (no more DELAF or DELACF).

NooJ can display dictionaries in a "list" form (like a DELAS), or in a table form (like a lexicon-grammar).

-- **Grammars** are used to represent a large gamut of linguistic phenomena, from the orthographical and the morphological levels, up to the syntagmatic and transformational syntactic levels.

In NooJ, there are different types of grammars. NooJ's three types of grammars are:

(a) Inflectional and derivational grammars (**`.nof`** files) are used to represent the inflection (e.g. conjugation) or the derivation (e.g. nominalization) properties of lexical entries. These descriptions can be entered either graphically or in the form of rules.

(b) Lexical, orthographical, morphological or terminological grammars (**`.nom`** files) are used to represent sets of wordforms, and associate them with lexical information, e.g. to standardize the spelling of word or term variants, to recognize and tag neologisms, to link synonymous expressions together;

(c) Syntactic, semantic and translation grammars (**`.nog`** files) are used to recognize and annotate expressions in texts, e.g. to tag noun phrases, certain syntactic constructs or idiomatic expressions, to extract certain expressions or interest (name of companies, expressions of dates, addresses, etc.), to disambiguate words by filtering out some lexical or syntactic annotations in the text or to perform transformations (e.g. automatic paraphrasing) or automatic translations (from one source language to another).

NooJ's Community

NooJ can be freely downloaded from www.nooj-association.org. Most laboratories and academic centers use NooJ as a research or educational tool: some users are interested by its Corpus processing functionalities (analysis of literary text, research and extract information from newspapers or technical corpora, etc.); others use NooJ to formalize certain linguistic phenomena (e.g. describe a language's morphology), others for computational applications (automatic text analysis), etc.

Visit NooJ's WEB site at www.nooj-association.org and the NooJ users' forum list at http://groups.yahoo.com/group/nooj-info to learn more about NooJ, its applications and its users.

Among NooJ users, some are actively helping the NooJ project, by giving away some of their linguistic resources, projects or demos, labs, tutorials or documentations. These users, who constitute "NooJ's community", should be considered as NooJ's "co-authors". The Community Edition of the NooJ application (which is also free), is an extended version of NooJ, that gives full access to its internal functionalities as well as privileged access to sources of its linguistic resources.

NooJ users meet once a year at the NooJ conference. NooJ tutorials and workshops are regularly organized during the year, some in French (e.g. "La semaine NooJ à l'INALCO"), some in English.

Structure of the book

This book is divided into five parts:

This section "**Getting Started**" presents NooJ (1), takes you through the installation process (2), and then helps you launch a basic search in a text (3).

The section "**Regular expressions and graphs**" shows you how to carry out simple searches in texts with regular expressions (4), how to use lexical resources for linguistic requests (5), and how to use NooJ's graph editor to describe more complex queries (6).

The section "**Text Processing**" explains how to import and process texts 7), and how to construct and process corpora (0). NooJ can import texts in over 100 file formats (all variants of ASCII, EBCDIC, ISO, Unicode, etc.), documents in a dozen formats (all variants of MS-WORD, HTML, RTF, etc.). NooJ can also process XML documents (see 0). In the latter case, XML tags can be imported into the Text's Annotation Structure, and a NooJ annotated text can be exported back to an XML document.

The section "**Lexical analysis**" describes NooJ dictionaries (9), NooJ's inflectional and derivational tools (10), and productive morphological grammars (11). NooJ uses these three tools to perform an automatic Lexical Parsing of texts (12).

The section "**Syntactic Analysis**" presents local grammars, and how to build libraries of graphs in order to build a bottom-up description of Natural languages using local grammars and remove lexical ambiguities (13). NooJ's parser's behavior is complex; we explain how it processes ambiguous and empty sequences, and how to set its priorities (14). Finally, we present more powerful grammars, such as enhanced RTNs used to perform automatic transformational analyses and translations (16).

The section "**References**" presents a number of tools aimed at teaching Linguistics, Corpus Linguistics and Computational Linguistics (17). We then describe every menu item and functionality (18). Finally, we present NooJ's standalone command-line program **noojapply.exe** that can be launched from a command-line "DOS" windows or a UNIX Shell environment. Chapter 19 contains the bibliography.

2. Installing the software

Installing NooJ

You can freely download NooJ from the NooJ Web site: www.nooj-association.org.

Just like any other software, it is important to make sure that your system has no virus nor any malware: more than one so-called "NooJ problem" has in fact turned out to be a consequence of some infected NooJ files...

Make sure to follow the instructions listed in the README.txt file in the directory.

NooJ's installation is straightforward and is based on the **XCOPY** model: simply copy NooJ's application .zip compressed folder on your computer, and then unzip it. You may delete the .zip file afterwards (Never run NooJ from the .zip compressed folder).

No "SETUP" or any modification of Windows' registry is necessary! Moreover, you do not need to have "Administrative" rights on a computer to install NooJ: you just need to be able to copy it, anywhere on the computer.

Go to NooJ's Web site's "Download" page, then download the version of NooJ you need: either the Windows version or the JAVA version.

-- Windows: click "Download NooJApp.zip (for Windows)". Uncompress it, for instance on your desktop, or into the usual application folder "c:\Program files", or into any other folder you wish. Please delete the .zip file afterwards to avoid running NooJ from the compressed folder (this will not work properly). In the resulting uncompressed folder, there is a "_App" folder; in this latter folder, locate the application file "NooJ.exe". You might want to make a shortcut to this file and store the shortcut either in Windows' "Start" menu, or on the desktop.

-- Java: click "Download NooJ – Java version (for Mac, LINUX and UNIX). Then double-click the Nooj.jar file that will launch the Java virtual machine and then NooJ.

Note for Mac users: you might get a warning message stating that the software has been downloaded from the Internet, because it was not acquired from the Apple store. In that case, right-click the Nooj.jar file, then click "Open with Jar launcher.app", then click "Open".

IMPORTANT: NooJ requires either the .NET (Windows) or the JAVA (Mac, Linux, Unix) virtual machines. Before proceeding any further, make sure that these virtual machines are already installed on your PC.

NooJ's files' extensions

If you wish, you can associate NooJ's files' types with the NooJ application, so that double-clicking these files will launch NooJ and open them automatically. The following file extensions can be associated with NooJ:

.DIC (dictionary)
.NOC (corpus)
.NOF (inflectional/derivational morphological grammar)
.NOG (syntactic grammar)
.NOM (productive morphological grammar)
.NOP (project)
.NOT (text)

Installing new modules for NooJ

The "NoojApp.zip" package contains two linguistic modules: the English and French standard modules. Members of NooJ's community have been uploading other modules for NooJ, including modules for Arabic, Western Armenian, Chinese, Hebrew, Italian, Quechua, Portuguese, Spanish, etc. Look at the "Resources" page at www.nooj-association.org to download these resources.

To add a module for NooJ, simply download the corresponding .zip file from the "Resources" page, then extract its content into the MyDocuments\NooJ folder, so that the new folder is at the same level as the standard "en" and "fr" folders. To be recognized as a valid language module, the new folder should contain the three sub-folders "Lexical Analysis", "Projects" and "Syntactic Analysis".

Registering NooJ's Community Edition

NooJ's standard edition does not require any registration and can be used freely. There is a community edition used by members of the NooJ Community, *i.e.* researchers who actively help NooJ's project by providing resources for NooJ. As NooJ's project is very ambitious (formalize any natural language, from the orthographic level up to the

transformational and semantic levels), there are many ways to help us! If you do wish to join NooJ's community, contact NooJ's author:

max.silberztein@univ-fcomte.fr

To run NooJ in the Community mode, go to the "**Info**" menu, click "**About NooJ**", select the "Community" option then enter your information (contact, institution, license key).

Personal folder

On the one hand, the NooJ software is stored in **NooJ's application folder**. On the other hand, **NooJ's personal folder** is the default folder in which NooJ stores all your personal data. NooJ's Windows version sets your NooJ personal folder to be located in your "My documents" folder:

```
My documents\NooJ
```

NooJ's Mac version sets your NooJ personal folder to be located in your "Documents" folder:

```
Documents/ONooJ
```

Your personal Windows settings may vary. For each language you are working with, NooJ creates one sub-folder (e.g. "en" for English, "fr" for French, etc.):

```
My documents\NooJ\en
My documents\NooJ\fr
```

in which it stores the corresponding linguistic resources. Each language folder in turn contains three embedded sub-folders: one to store lexical resources, one to store syntactic and semantic resources, and one to store corpora, projects and texts, e.g.:

```
My documents\NooJ\en\Lexical Analysis
My documents\NooJ\en\Projects
My documents\NooJ\en\Syntactic Analysis
```

Preferences

NooJ's behavior follows a number of default parameters that are set via the **Info > Preferences** control panel. There, you can set your default working language, default fonts to display texts and dictionaries, what lexical and syntactic resources are applied for each language, etc.

Updates

NooJ's computational functionalities and linguistic resources are updated regularly. To upgrade the software, just replace the NooJ application folder (the one you might have stored on your desktop, or in `C:\Program files`), with the latest version available at the NooJ web site:

www.nooj-association.org

Your own data should be stored in your personal folder, by default: "My Documents\Nooj". **Never, ever** store any of your data in the NooJ application folder, as it might be lost at the next update.

All the files with names that start with the prefix "_" (underscore character) might be updated or deleted at each new update cycle. For this reason, never save any file with the prefix "_". If you do want to edit such a file, make sure to rename it when you save it (**File > Save As**) so that it does not get destroyed at the next NooJ update.

The nooj-info forum

Check out the NooJ forum regularly at:

http://groups.yahoo.com/group/nooj-info

to be kept informed about major updates, availability of new modules, as well as FAQs, tips, etc.

If you subscribe to this group, make sure to set the list to automatically send any messages to **your regular email address** rather than the new Yahoo email address that Yahoo gives you when you register.

NooJ's Web site contains a History page, available from the **download** page, in which important new functionalities are regularly described.

Uninstalling NooJ

If you wish to uninstall the software, simply delete the NooJ application folder (i.e. the one you created when you installed the software).

Note that your work and personal data and parameter settings have been stored in your user folder (**not** in the NooJ system folder) therefore they will not be deleted. If you want to delete all NooJ's linguistic resources as well, you can delete the user folder (usually in `My Documents\NooJ`).

3. Quick start

First, let's learn how to use one of NooJ's most basic functions: the ability to locate words and expressions in a text.

Loading a text

If you have not yet done so during the installation process, make sure to create a short cut, on your desktop or in your Start menu, to the file "**`Nooj.exe`**" located in NooJ's "_App" folder, e.g.:

```
c:\Program files\NooJ\_App\Nooj.exe
```

On a Mac, LINUX or UNIX PC, Launch NooJ by double-clicking the Nooj.jar file.

Now click the menu items: **File > Open > Text**. You should see a few text files, with the extension ".NOT" (for "NooJ Text"). Select the file "**_The Portrait Of A Lady.not**" (the novel by Henry James). The text will load and you should see a window like the one below:

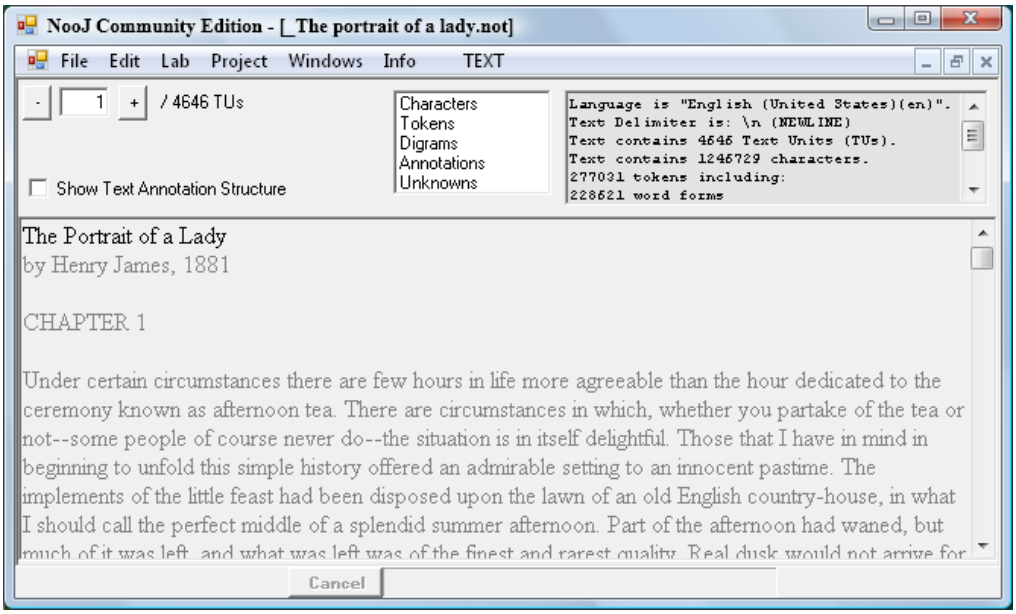


Figure 1. Loading the text "The Portrait of a lady"

At this point, default linguistic resources have already been applied to the text. NooJ produces some indications, displayed above and to the right of the text window:

```
Language is "English (United States)(en)".
Text Delimiter is: "\n" (NEW LINE)
Text contains 4646 Text Units (TUs).
285993 tokens including:
233102 wordforms
1249 digits
527659 delimiters
Text contains 1013374 annotations.
```

First of all, a few definitions:

Letters are the elements of the alphabet of the current language. Digits are the ten digit characters (from "0" to "9"). The Blank in NooJ represents any sequence of spaces, tabulation characters, NEWLINE and CARRIAGE RETURN. Delimiters are all the other characters.

From these definitions, NooJ uses the following definitions:

Tokens are the basic linguistic objects processed by NooJ. They are classified into three types: Wordforms are sequences of letters between two delimiters; Digits; and Delimiters. Digrams are pairs of wordforms (we ignore the delimiters between them).

Note that NooJ processes digits and delimiters both as characters and as tokens.

When processing certain Asian languages, NooJ processes individual letters (rather than sequences of letters), as tokens.

Some unusual examples:

For NooJ, the sequence "o'clock" is constituted of three tokens: the **simple form** "o", followed by the **delimiter** " ' ", followed by the **simple form** "clock". Similarly the adverb "**a priori**" is made up of two tokens (blanks do not count).

The sequence "3.14" is made up of one **digit**, one **delimiter**, and then two **digits**, that make four tokens. The sequence "PDP/11" is made up of the **simple form** "PDP", followed by the **delimiter** "/", followed by two **digits** (which make four **tokens**).

Our text has 233,102 wordforms, 1,249 digits (that low number is characteristic for litterary texts) and 52,765 delimiters (i.e. roughly one punctuation character every 5 wordforms).

Double click in the Results window (the little area above the text), to display the lists of the text's **Characters**, **Tokens** and **Digrams**:

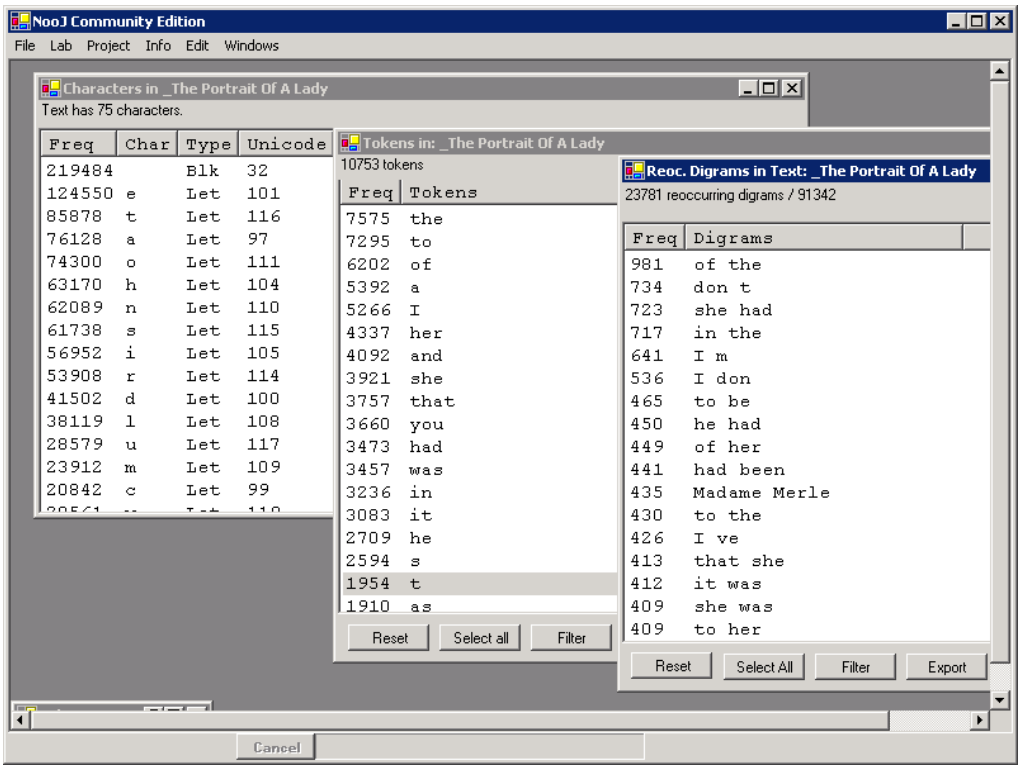


Figure 2. Several results: the text's characters, tokens and digrams

The most frequent characters are the space character (it occurs 219,484 times in this text) and the letter "e" (it occurs 124,550 times).

The most frequent tokens in this text are the wordform "the" (7,575 times), and "to" (7,295 times).

The most frequent digrams are the sequence "of the" (981 times) and "don t" (734 times). The digram "don t" corresponds to the sequence with the apostrophe "don't": NooJ does not compute the two digrams (don, ') and (',t). Hapaxes, i.e. digrams that only occur once, are not displayed.

Digrams are sequences of wordforms, i.e. delimiters are simply ignored in digrams.

The three lists (characters, tokens and digrams) can be sorted alphabetically, from left to right, or from right to left, or according to the frequency of each item.

Based on the previous definitions of characters and tokens, NooJ defines **Atomic Linguistic Units (ALUs)**.

NooJ processes four types of Atomic Linguistic units:
-- Affixes (prefix, proper affix or suffix) are the smallest sequences of letters included in wordforms that must be associated with relevant linguistic data, e.g. *re-*, *-ization*. In NooJ, they are described by inflectional rules, derivational rules or (productive) morphological grammars;
-- simple words are wordforms that are associated with relevant linguistic information, e.g. *table*. They are usually described in dictionaries;
-- multi-word units are sequences of tokens (wordforms, blanks, delimiters and/or digits) associated with relevant linguistic information, e.g. *as a matter of fact*. They are usually described in dictionaries;
-- expressions are potentially discontinuous sequences of wordforms that are associated with relevant linguistic information, e.g. *take ... into account* . They are described either in dictionaries or in syntactic grammars.

Do not confuse the terms "wordforms" (a type of token) and "simple words" (a type of atomic linguistic unit represented in a NooJ linguistic resource).

For instance, the two wordforms "THE" and "the" are different **tokens**. They are usually associated with a unique Atomic Linguistic Unit (**simple word**) because there is only one lexical entry "the = determiner" in NooJ's dictionaries that matches both wordforms.

However, the single French wordform "PIERRE" might correspond to two different ALUs (simple words) because NooJ can link the token to two dictionary entries "pierre" (noun meaning "stone"), and "Pierre" (a French firstname).

Note that the wordform "pierre" will be linked to the only dictionary entry "pierre,N", not to the firstname which must be written in uppercase. See NooJ's case conventions in the dictionary section.

NooJ has inserted annotations in this Text's Annotation Structure:

```
Text contains 809111 annotations.
```

An annotation is a pair (*position*, *information*) that states that a certain sequence in the text (located at a certain position) is associated with some information. Annotations can be added to the Text Annotation Structure by three mechanisms:

-- NooJ's lexical parser adds four types of annotations to the text, corresponding to the four types of Atomic Linguistic Units (affixes, simple words, multi-word units and expressions).

-- NooJ's syntactic parser also can add annotations to, or remove annotations from, the Text Annotation Structure.

-- NooJ can process XML documents, in which case certain XML tags can be imported as annotations into the Text Annotation Structure.

Note that at this stage, NooJ's lexical parser has produced a high level of ambiguity (233,000 wordforms produce 809,000 annotations), typical for English texts. We will need a good syntactic component to lower the level of ambiguities.

Let's look at the results of the lexical analysis: above the text window, double-click the "**Annotations**" (to display the information that is being associated to the text), and then the "**Unknowns**" results (to display the wordforms that have not been associated with any annotations):

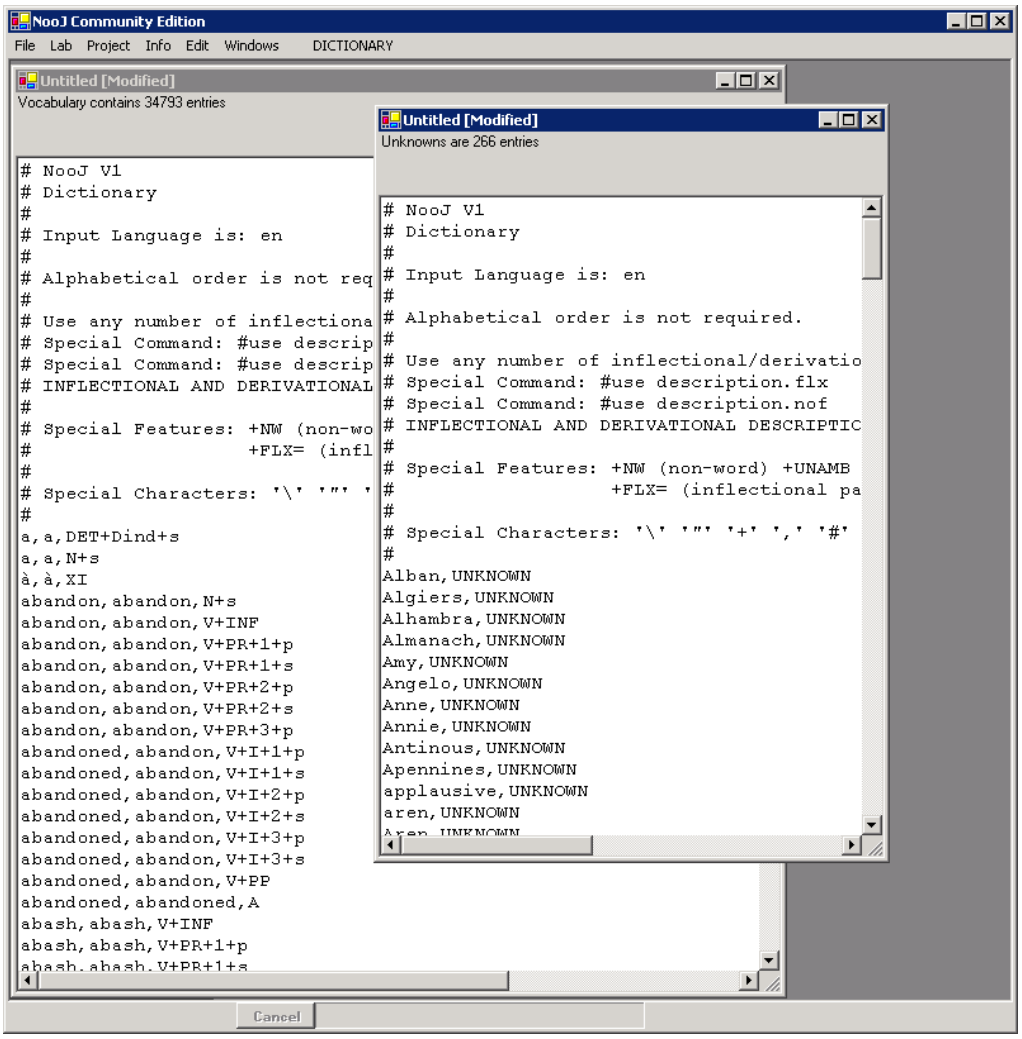


Figure 3. Lexemes and Unknowns

NooJ displays these two lists in the native NooJ dictionary format. These two windows can be edited, typically, in order to replace the code "UNKNOWN" with something more useful.

We will see later the signification of the codes.

Locating a wordform

In the **TEXT** menu, click "**Locate**". The "**Locate Panel**" window will show up. In the field "**Pattern is:**", select the option "**a NooJ regular expression:**" (you will enter a regular expression), then type "`perhaps`" in the field (A). Then click a colored button in the lower right corner (B) of the window, for instance the red one. The search operation is launched.

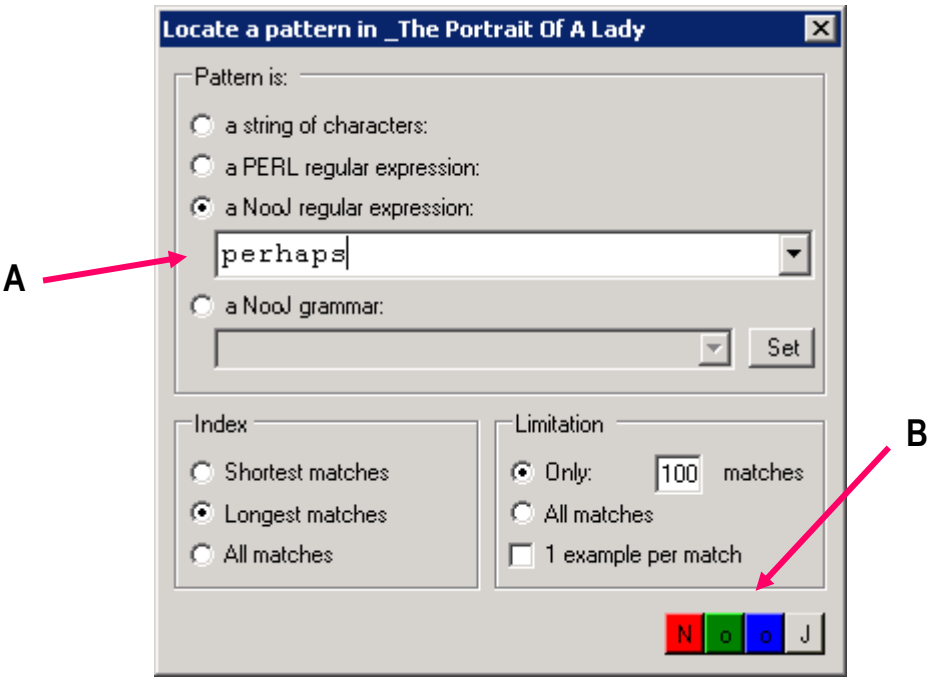


Figure 4. Locate a word

NooJ lets you know that it found 100 matches for your query, and then displays a concordance in the selected color.

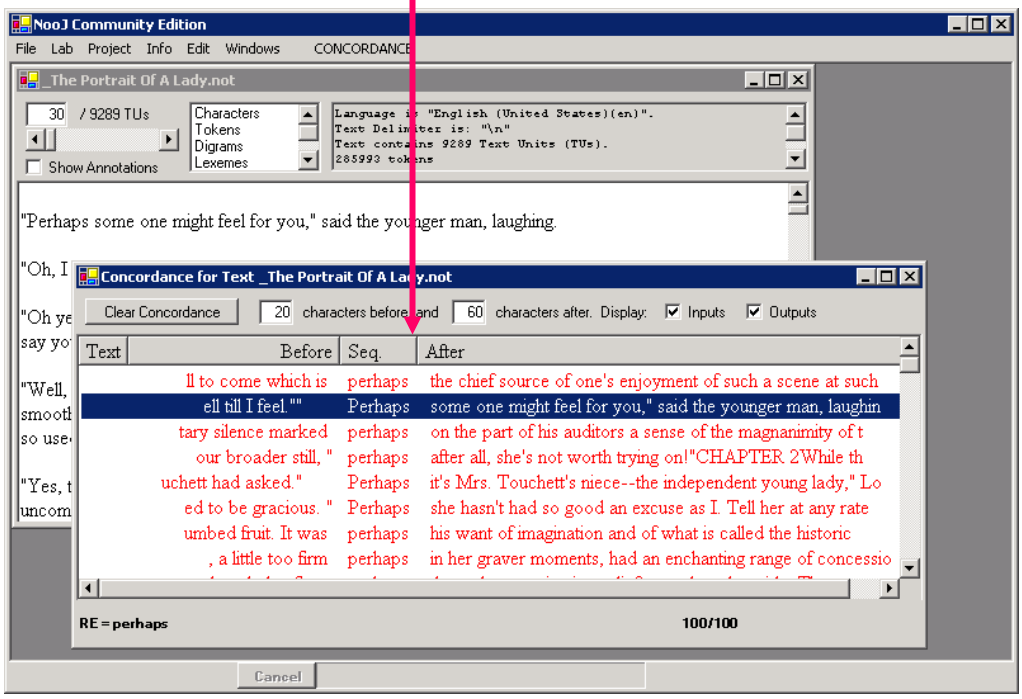


Figure 5. Concordance of the word "perhaps"

Double-clicking one entry of the concordance makes NooJ display the corresponding matching occurrence within the text.

The concordance of a sequence is an index that represents all of its utterances in context. NooJ concordances are displayed in four columns: each occurrence being presented in the middle column, between its left and its right context. If a corpus (i.e. a set of text files, rather than a single text) is being indexed, the first column displays the text file name in which each match occurs.

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, NooJ displays the matching occurrence within the text. Note that clicking the header of the "Before" context makes NooJ sort the concordance from the end of the preceding wordforms.

REGULAR EXPRESSIONS AND GRAMMARS

The second part shows you how to carry out complex searches in texts with regular expressions (chap. 4), how to use lexical resources for linguistic queries (chap. 5), and how to use NooJ's grammar editor to describe more powerful queries (chap. 6).

4. Regular expressions

Disjunction

Load the text file "_Portrait of a lady" (**File > Open > Text**). Display the locate window (**Text > Locate**). Now type in the following **NooJ regular expression** (spaces are optional):

```
never | perhaps
```

In NooJ, the disjunction operator (also known as the UNION, or the "or") is symbolized by the "|" character.[1]

The disjunction operator, introduced in NooJ as the character "|", tells NooJ to locate all of the utterances for "never" or "perhaps" in the text.

Make sure there is no limitation to the search: select "**All matches**" in the **Limitation** zone (**A**), 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 "**Only 100 matches**", the search would have been limited to the first 100 matches.

[1] NooJ accepts the operator "+" also to denote the disjunction. In this manual, we will only use the "|" operator, since the "+" operator is ambiguous as it is also used as a prefix for lexical properties.

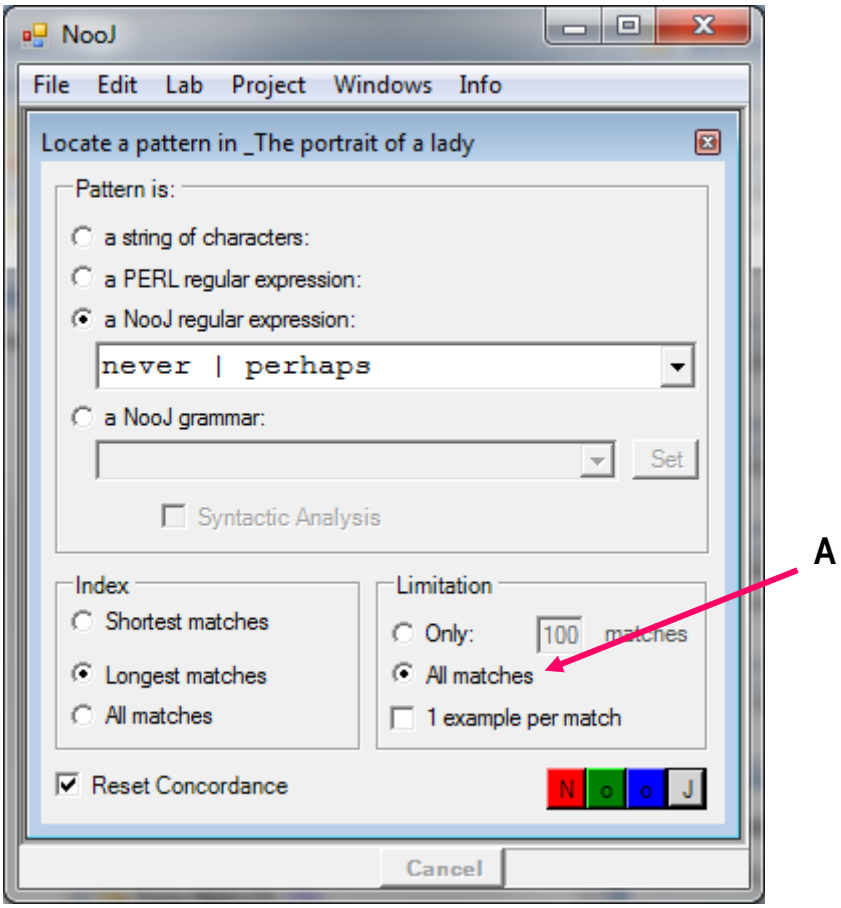


Figure 6. Enter a NooJ regular expression

Now click one of the colored buttons at the bottom of the **Locate** panel: the search is launched. After a few seconds, you should get a concordance with 696 entries. Click on the "**After**" column header in order to sort the concordance entries according to their right context (if you are interested by what occurs after these adverbs).

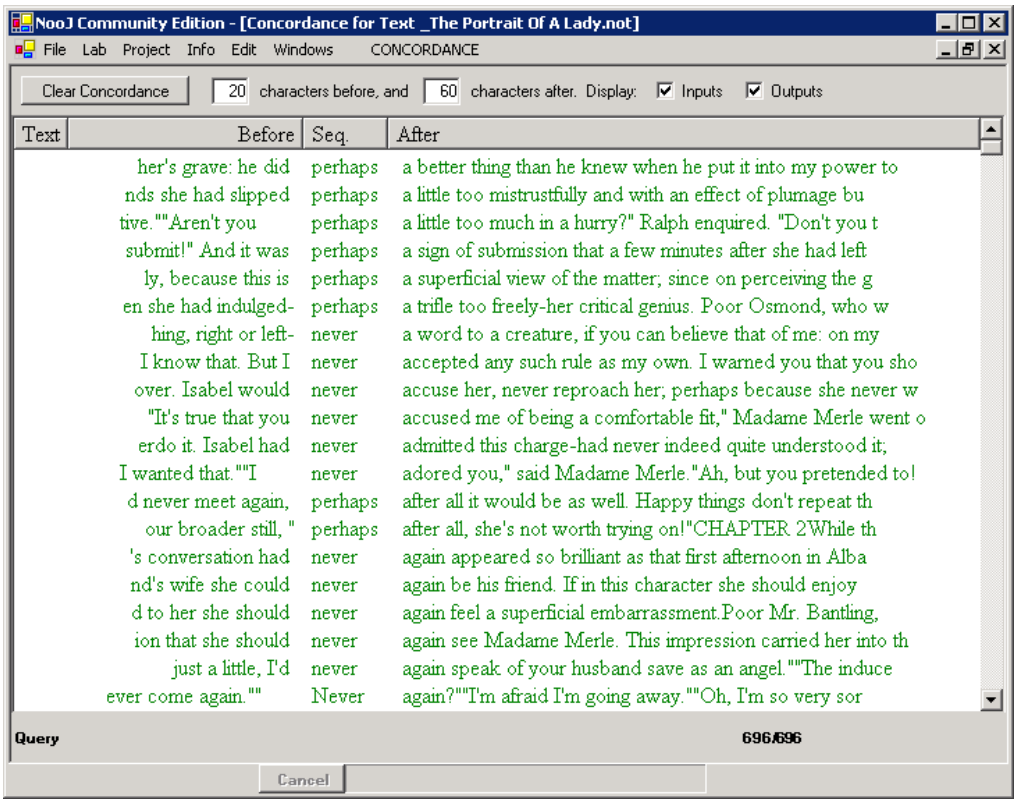


Figure 7. Concordance for the expression: never|perhaps

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**), then enter the following regular expression:

```
(her | his) voice
```

Click on a colored button (not the color you already selected for the previous concordance). NooJ should find 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, NooJ recognized 4,495 utterances:

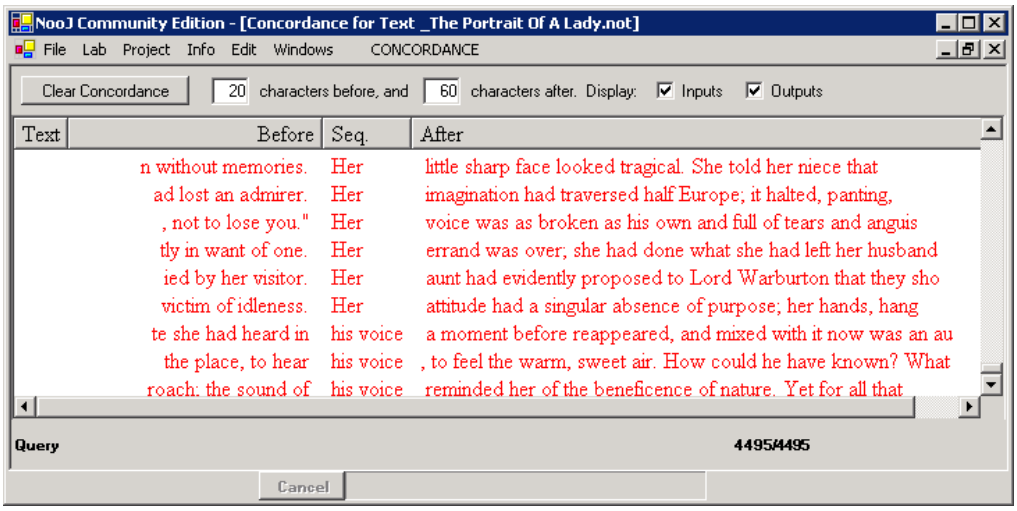


Figure 8. Forgotten parentheses

What happened? NooJ 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.

Sets of forms

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

```
am|are|is|was|were
```

In the lower left-hand corner (B), under **Limitation,** make sure the radio button "**All matches**" is selected, then click on a colored button to launch the search.

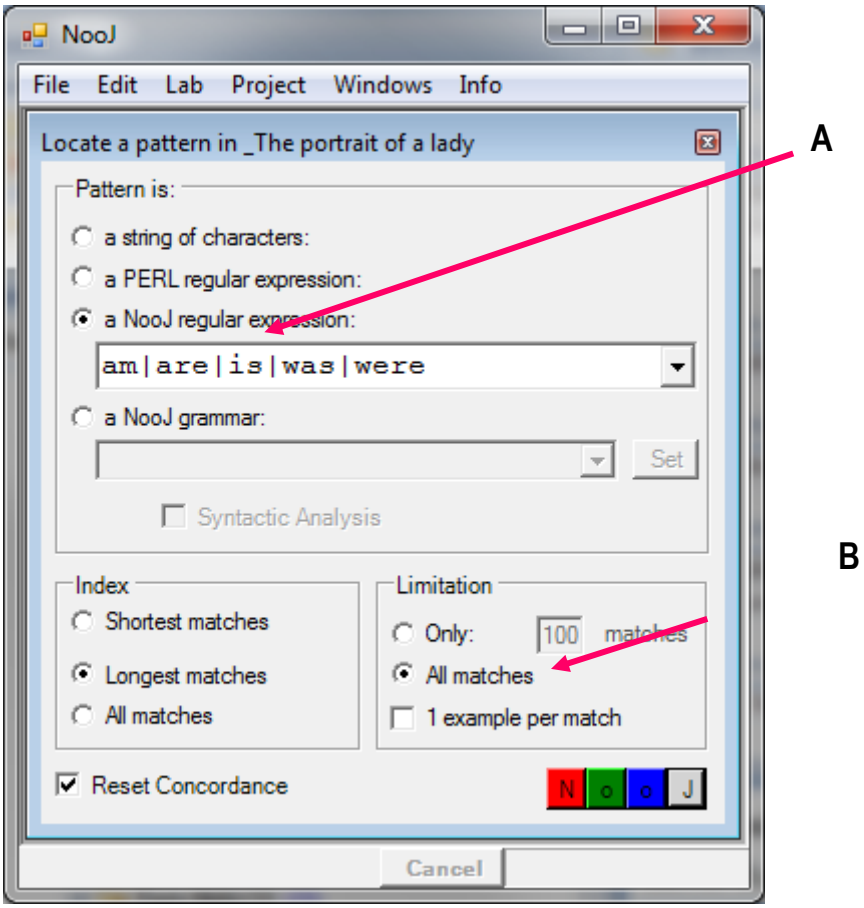


Figure 9. Locate a set of forms

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.

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 wordforms:

```
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".

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 multi-word units does this wordform occurs?

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

Study the use of the wordforms *up* and *down*: how many times these wordforms correspond to a particle; how many times do they correspond to a preposition?

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

Special symbols

The following regular expression:

```
(the | a) <WF> is
```

finds all of the sequences made up of the wordform "the" or "a", followed by any wordform (<**WF**>), followed by the form "is".

All NooJ special symbols are written between angle brackets "<" and ">". Do not leave any blank space between the angle brackets and respect the case (upper-case for the symbol <**WF**>). If you apply exactly the former expression to the text, you should obtain a concordance that looks like the one below.

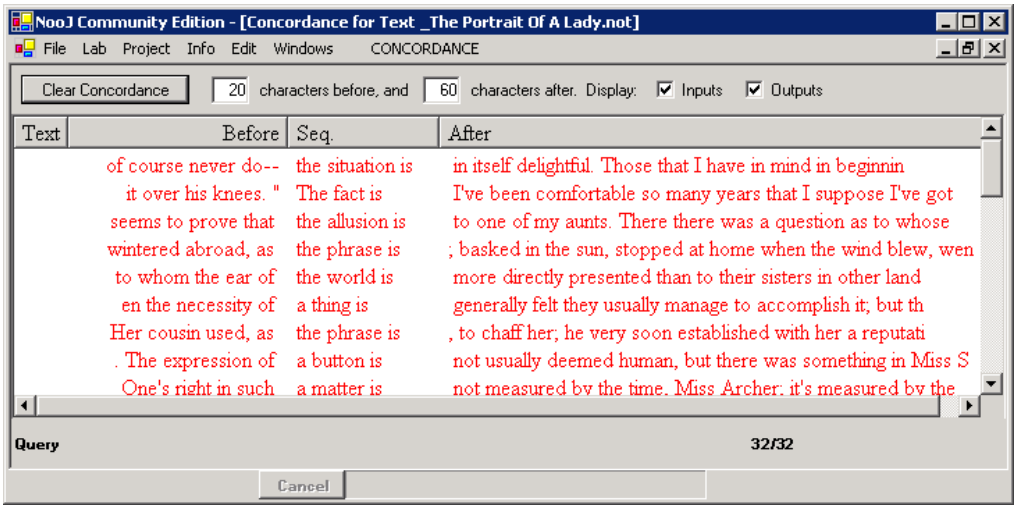


Figure 10. Apply a query with a special symbol

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

```
(the | a) WF is
```

represents the two literal sequences "the WF is" and "a WF is". There isn't much chance of you finding that sequence in this text…

IMPORTANT: <WF> is a special symbol. In NooJ, all symbols are written between angle brackets.

Following is a list of NooJ symbols, as well as their meaning:

Special Symbol	Meaning
<WF>	*wordform (Sequence of letters)*
<L>[2]	*wordform with length 1*
<LOW>	*wordform in lower-case (sequence of lower-case letters)*
<W>	*wordform in lower-case, with length 1*
<UPP>	*wordform in upper-case (sequence of upper-case letters)*
<U>	*wordform in upper-case, with length 1*
<CAP>	*wordform in capital (an upper-case letter followed by lower-case letters)*
<NB>	*sequence of digits*
<D>	*one digit*
<P>	*delimiter (one character)*
<^>[3]	*Beginning of a text unit*
<$>	*End of a text unit*
<V>[4]	*a vowel*

Following are a few expressions that contain special symbols:

We are looking for all the paragraphs that start with a wordform in capital, followed with a form in lower-case, and then a colon:

```
<^> <CAP> <LOW> ,
```

Apply this query to the text "The portrait of a lady": you should get 155 matches.

Now we want to locate the wordforms in upper-case that occur at the beginning of paragraphs, or after a comma, and are followed by the wordform "said":

```
(<^> | ,) <CAP> said
```

---

[2]. At the syntactic level (i.e. from the **Locate** panel) the symbols **<L>**, **<U>** and **<W>** match wordforms of length 1. At the morphological level, they match single letters inside wordforms.

[3]. Text units are defined by the user, when the text or corpus is opened: they can be paragraphs (by default), sequences of the text delimited by a certain PERL pattern, a text zone delimited by an XML tag, etc. At the morphological level, linguistic units are wordforms.

[4] This symbol is used with this meaning only at the morphological level. At the syntactic level, this symbol is not a special NooJ symbol. We will see that it gets another meaning (usually "verb").

(there are 6 occurrences in the text). Now we want to locate all sequences of two consecutive forms written in upper-case letters:

```
<UPP> <UPP>
```

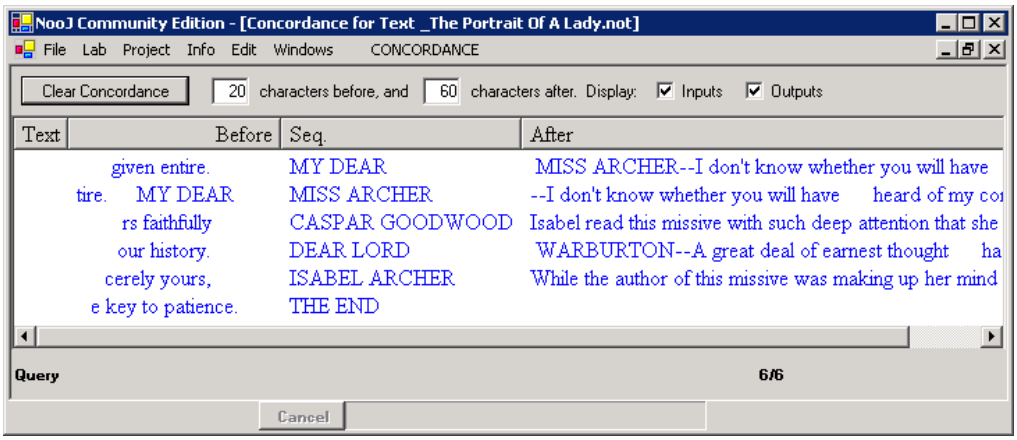


Figure 11. Search for a sequence of two upper-case forms

Now locate the wordforms that occur between "at" or "in" and "of" (there should be 142 matches):

```
(at | in) <WF> of
```

Special characters

The blank

NooJ processes any sequence of spaces, tabulation characters, and line change (codes "NEW LINE and "CAR RET") characters as **blanks**. When entering a regular expression, blanks are usually irrelevant and are therefore optional.

Generally, one does not search for blanks:

In morphology, the range of the search is limited to the wordform, in which there is no blank, by definition; in syntax, blanks are always implicit; the expression <WF><WF>, for example, recognizes any sequence of two wordforms (that are naturally separated by space characters in languages such as English). 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 NooJ 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          ,
```

Double quotes

It is sometimes necessary to specify "mandatory" blanks; in which case, we can use the double quotes to make the space explicit. 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.

More generally, double quotes are used in NooJ 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 wordforms in parentheses, we would enter the expression:

```
"(" <WF> ")"
```

Similarly, if we want to locate uses of the character "|" between numbers:

```
<NB> "|" <NB>
```

Double quotes are not useful in the following case: the expression in the second line is simpler and equivalent to the first line:

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

Double quotes 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 wordform *is*, 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>
```

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 anything.

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) | <WF> (the|this|that) |
          <WF> <WF> (the|this|that))
```

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

```
is (<E> | <WF> | <WF> <WF>) (the|this|that)
```

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 wordforms, followed by the wordform "the", the following expression would be used:

```
is <WF>* the
```

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

```
is (<E> | <WF> | <WF> | <WF><WF> | ...) the
```

In the same manner, 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, the very 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 wordform "is", then of any possible insertion, then of the wordform "by", you should enter the expression:

```
is (<WF> | <P>)* by
```

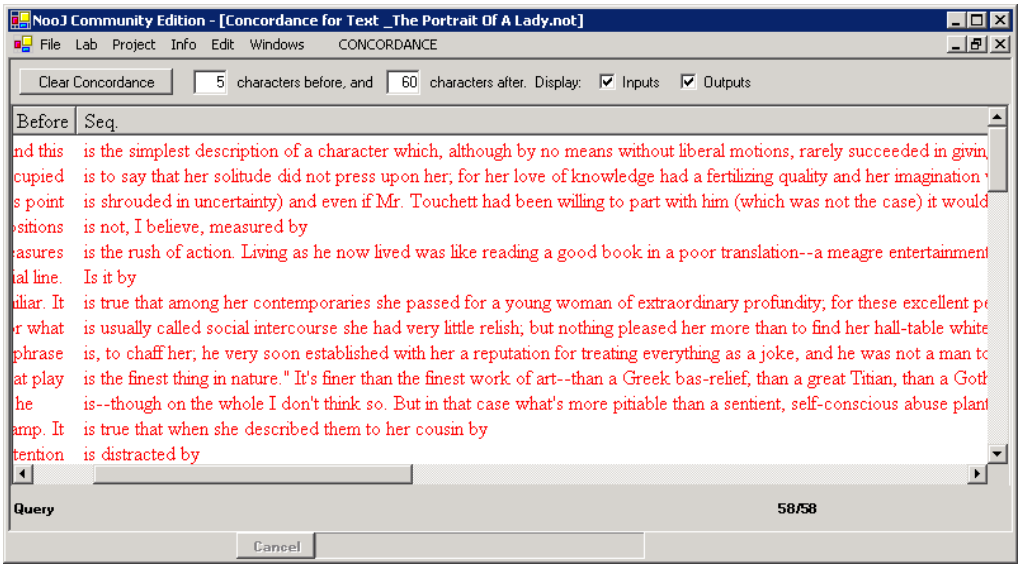


Figure 12. Arbitrary sequences in a pattern

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

We will see later that regular expressions are also used in NooJ's inflectional and derivational morphology module.

Summary:

You have learned to write a few elementary syntactic 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 unlimited 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.

5. Using lexical resources in Grammars

Indexing all inflected forms of a word

Previously, we located the conjugated forms of the verb *to be* by using the following expression:

```
am | are | is | was | were
```

We could also add the following forms to the expression:

```
be | been | being
```

While the resulting expression would be perfectly valid, that would certainly be very tedious. Fortunately, it is possible to use lexical information to greatly simplify this type of queries.

For each language, NooJ accesses a dictionary (further described later) in which each word of that language is an entry, and is associated to some morphological information, usually, its inflectional and/or derivational paradigms. The inflectional paradigm tells NooJ what *inflected forms* the lexical entry accepts, i.e. what are its conjugated forms (if it is a verb), its feminine and plural forms (for nouns in Romance languages), its accusative, dative, genitive etc. forms (for Germanic languages), etc. The derivational paradigm tells NooJ what derived forms the lexical entry, i.e. what word can be derived from the entry. For instance, from the noun "color", we can construct the verb "to colorize", from the verb "to drink", we can construct the adjective "drinkable", etc.

Thanks to this dictionary, NooJ can link all inflected or derived forms together. All these forms are then stored in an equivalence set. We access this equivalent set simply

by entering any member of the set between angle brackets. For instance, the following expression, in which we refer to the wordform "be", represents all of the inflected forms of "to be", in which we are interested.

```
<be>
```

Load the text "_Portrait of a lady" (**File > Open > Text**). Display the locate window (**Text > Locate**). Enter this regular expression in the text's "Locate" panel. 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 *be* in lower-case. Apply this expression without any limitations to the text "The Portrait Of A Lady". NooJ should find 7,484 utterances. In the same manner, re-launch the search by using the symbol:

```
<was>
```

NooJ should find the same 7,484 utterances, i.e. exactly like previously because the two wordforms "be" and "was" belong to the same equivalence set.

Always remember to write the angles! re-launch the search but without typing in the angles:

```
be
```

This time, NooJ only locates the utterances for the literal wordform "be" (only 1,366 occurrences).

In a regular expression, when a form is written as is (e.g. be), NooJ locates the utterances of the wordform itself. On the other hand, when the wordform is set between angle brackets, NooJ locates all of the wordforms that are in the same equivalence set as the given wordform (generally all inflected, derived forms or spelling variants of a given lexical entry).

1. Indexing a category

In NooJ's dictionaries, all entries are associated to a morpho-syntactic category. 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>
```

(In NooJ's English dictionary, **PREP** stands for **Preposition**, and **N** denotes **Noun**). Launch the search; NooJ should show 373 sequences.

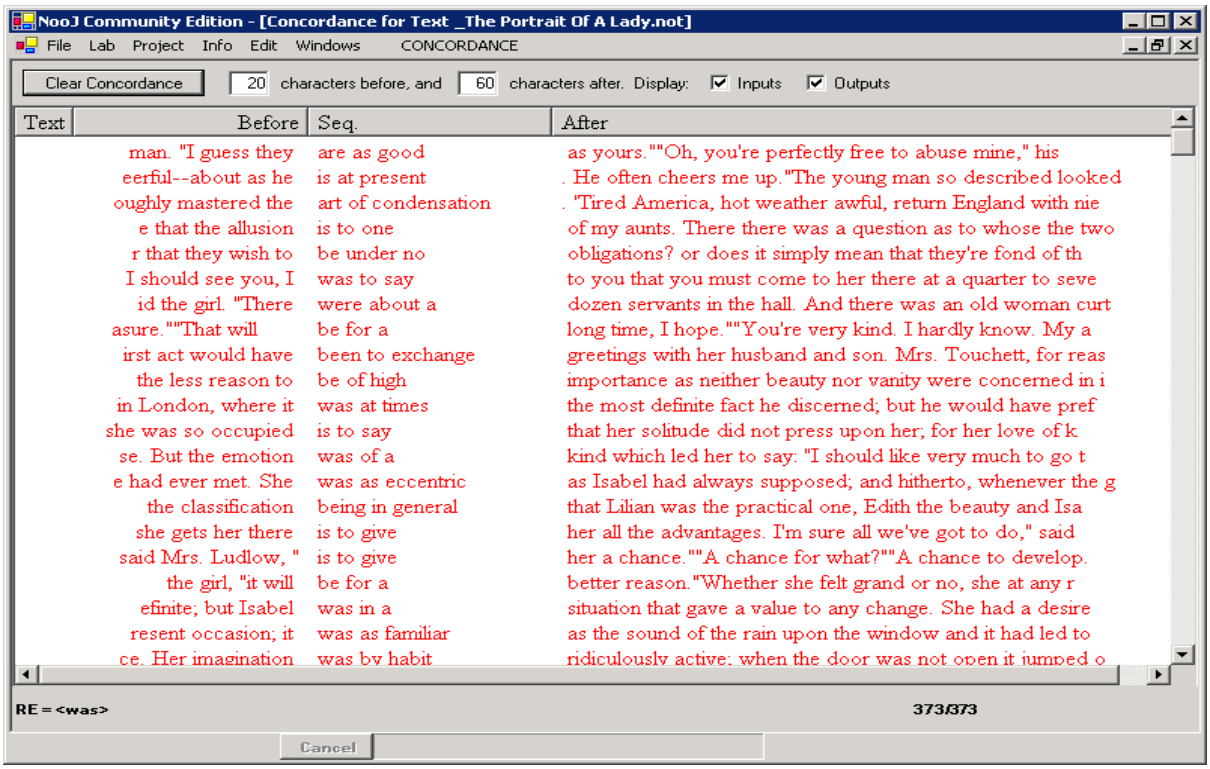


Figure 13. Use lexical information in regular expressions

NooJ will locate all of the sequences made up of any wordform in the same equivalence set as "be", followed by any wordform associated with the "PREP" category, followed by any wordform associated with the "N" category.

The following symbols are references for the codes found in NooJ's English dictionary:

 **Code**	**Meaning**	**Examples**
ALU	*Atomic Linguistic Unit*	*any textual unit described by a dictionary or a morphological grammar*
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*

Note: these codes are not set by NooJ itself, but rather by its dictionary. NooJ does not know what the symbol "ADV" means: in order to recognize the special symbol <ADV>, NooJ consults its dictionaries, and verifies if the word is therein associated with the code ADV.

In other words, linguists and lexicographers who are using NooJ are totally free to invent their own category and codes (e.g. **DATE**, **VIRUS** or **POLITICS**).

Important: Users may add their own 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 available for any query, and may be instantly used in any regular expression or grammar (just write them between angle brackets).

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 professions with the code <PRO> if you plan to use NooJ's default dictionaries, because this code is already used for 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 queries and grammars you might want to write may access all nouns with one single symbol.

We will now locate the sequences of the verb "to 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
```

Launch the search; NooJ should find the corresponding sequences.

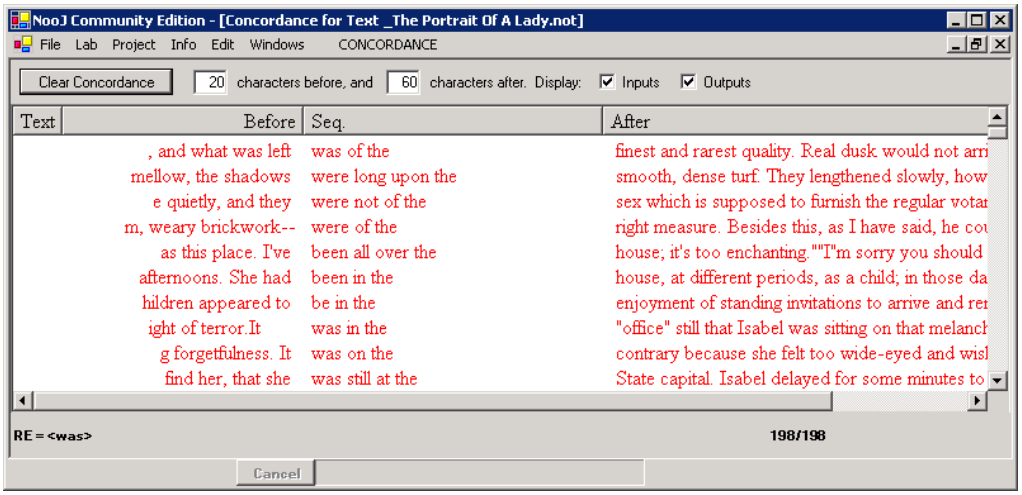


Figure 14. Another regular expression

Combining lexical information in symbols

In NooJ's dictionaries, entries are associated with at least one morpho-syntactic code. They may also be described with other types of information, and all of the information available in these dictionaries may be used in queries or in grammars. For example, here is an entry from the English dictionary:

```
virus,N+Conc+Medic
```

This entry states that the word "virus" is a noun (N), belonging to the distributional class **Concrete** (Conc) and is used in the semantic domain **Medical** (Medic).

Syntactic and semantic information

All pieces of information in NooJ are represented by codes prefixed with the character "+".

Warning: do not confuse the "+" character in dictionaries and the disjunction operator "|" in regular expressions.

One can use these codes in queries, to the right of a wordform or of a category. For example, <fly+tr> could denote transitive uses of the verb *to fly*, and <N+Medic> represents all the nouns that are associated with the medical semantic domain.

Symbols in queries can include negations. For instance, <fly-tr> would denote non-transitive uses of the verb to fly, and <N-Medic> represents all the nouns that are not associated with the medical semantic domain.

One can combine these codes as much as needed. For example <N+Hum-Pol> represents human (+Hum) nouns that do not belong to the semantic domain "Politics". Codes are not ordered: for instance, the previous symbol is equivalent to <N-Pol+Hum>.

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

Inflectional Information

In NooJ, any piece of lexical information (including inflectional codes) is encoded the same way, i.e. with the prefix character "+". However, inflectional codes are not, usually, visible in NooJ's dictionaries, because NooJ's dictionaries contain lemmas, rather than conjugated forms. We will see later that inflectional codes are described in the inflectional-derivational description files (.nof files).

However, it is important to know what these codes are, because they can be used (questioned) exactly as syntactic or semantic codes. Here are the inflectional codes that are used in NooJ's English dictionary:

 Code	**Signification**
s	*Singular*
p	*Plural*
1, 2, 3	*1st, 2nd, 3rd person*
PR	*Present tense*
PRT	*Preterit*
PP	*Past participle*
G	*Gerundive*
INF	*Infinitive*

Both in symbols and in dictionaries, information codes are not ordered. For example <V+PR+3+s> and <V+3+PR+s> match the same utterances. NooJ allows any partial queries, for example, <be+PR> represents all of the forms of the verb *to be* conjugated in the Present tense, and <be+3+s> matches both forms "is" and "was". Below are the inflectional codes that are used in NooJ's French dictionary:

 Code	**Signification**
s	*Singulier*
p	*Pluriel*
1, 2, 3	*1ère, 2ème, 3ème personne*
PR	*Présent de l'indicatif*
F	*Future*
I	*Imparfait*
PS	*Passé simple*
S	*Subjonctif présent*
IP	*Impératif présent*
C	*Conditionnel présent*
PP	*Participe passé*
G	*Participe présent*
INF	*Infinitif*

It is possible to combine queries on a word and on a category: for example, the symbol <admit,V+PR> matches all the forms of the verb "admit" conjugated in the present. We will see that these complex queries are useful when a text has been partially (or totally) disambiguated.

Negation

NooJ processes two levels of negation in symbols used in regular expressions:

-- as we have just seen, one can prefix any of the properties of a lexical entry with the character "−" instead of the character "+"; in that case, only

wordforms that are *not* associated with the feature will match; for instance, <N-Hum> matches non-human nouns.

-- another, more global negation: one can match all the wordforms that do **not** match a given lexical symbol, by prefixing the symbol with the character "!". For instance, <!V> matches all the wordforms that are not annotated as verbs; <!have> matches all the wordforms that are not annotated with the lemma "have"; <!N+Hum+p> matches all the wordforms that are not annotated as plural human noun.

Warning: negations often appear to produce obscure, unexpected results in NooJ, because of the huge level of ambiguity that is produced by dictionaries.

For instance, consider the following untagged text:

I left his address on the table

the query <!V> would match all the wordforms in the previous sentence, including "left", because this form is also associated with the lexical entry left = Adjective, therefore "left" *can be* a non-verb.

<!N> also matches all the forms, including "address" and "table", because both forms are also associated with lexical entries that are *not* nouns (the verbs "to address" and "to table").

As a consequence, I strongly suggest limiting the use of the negation in queries that are applied to texts after they have been disambiguated. We will see later how to perform disambiguation with NooJ. As a matter of fact, all these problems disappear if one works with the same text, after it has been tagged the following way:

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

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

PERL-type Expressions

It is also possible to apply a SED-GREP-PERL-type regular expression to constraint the wordforms that match a certain pattern. To do so, use the +MP="..." feature. For instance:

```
<ADV+MP="ly$">
```

matches all the adverbs ("ADV") that end with "ly" (in the PERL pattern: "ly$", the "$" character represents the end of the wordform). In the same manner:

```
<UNK-MP="^[A-Z]">
```

matches all the unknown words ("UNK") that do not start with an uppercase letter (the PERL special character "^" represents the beginning of the wordform; the set [A-Z] represents any of the characters A, B, ... Z). Finally, it is possible to combine more than one PERL-type matching pattern:

```
<N+Hum-MP="or$"-MP="[Aa]">
```

human nouns that do not end with "or", and do not contain any "a" or "A".

Symbols in regular expressions represent:
-- wordforms characterized by their case; e.g. <LOW> matches all lowercase wordforms;
-- wordforms that belong to a morphological equivalence set; e.g. <have> matches all forms of the verb "to have";
-- wordforms associated with a morpho-syntactic category; e.g. <PREP> matches all prepositions;
-- any number of the codes that are available in NooJ's dictionary can be used in combinations; e.g. <N+HUM+Medic+p> (plural human noun of Medical vocabulary) or <V+tr+Pr-3> (any transitive verb conjugated in the Present and not in the third person);
-- negations can be used either globally (with the "!" character) or for each property (with the "-" character).

NooJ Grammars

NooJ allows users to save queries in **grammars**. A grammar is a collection of one or more queries. Each query is given a name, and must be followed by a semi-colon, such as in:

```
myquery = <V> <PREP> <N+s> ;
```

In order to create a grammar file, use **File > New > Grammar**. Then select the language (e.g. "en"), the format "rule editor", and the click the "Syntax" button:

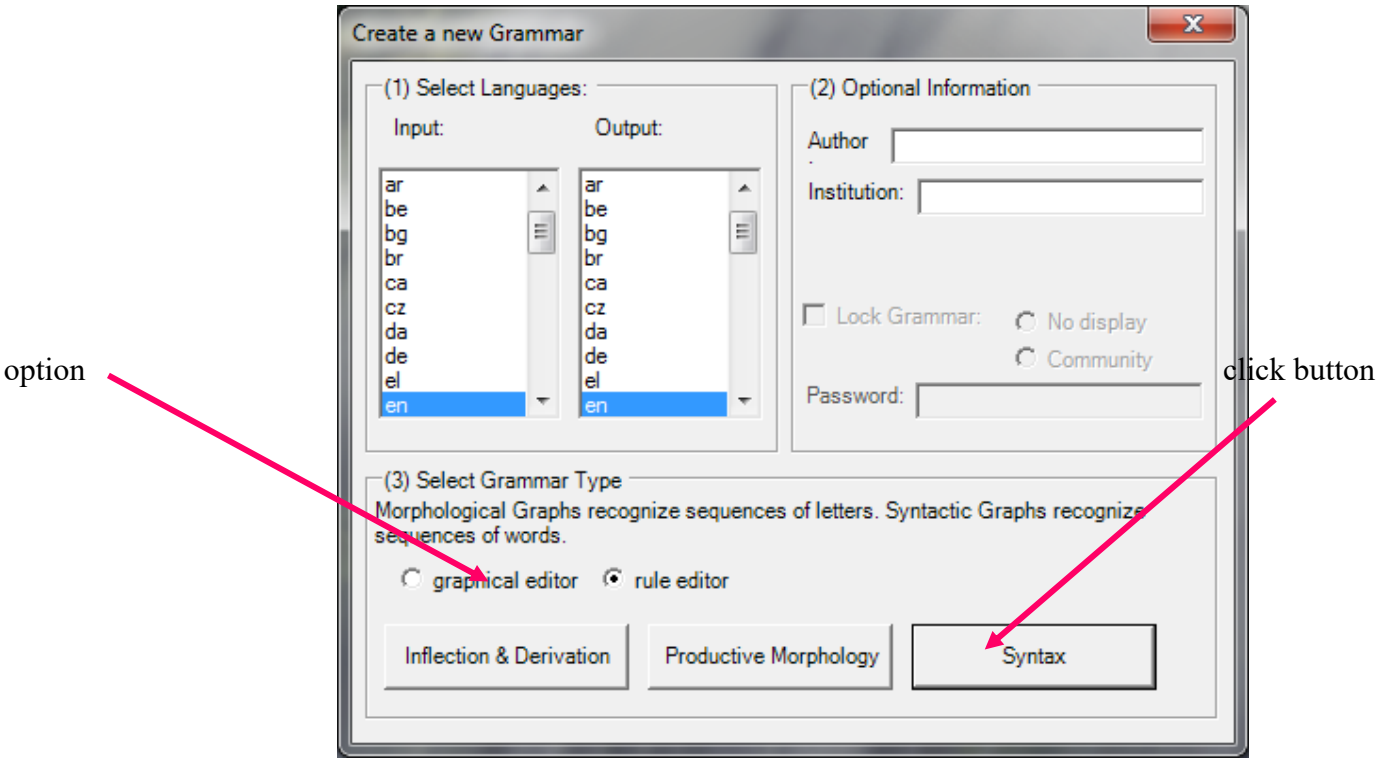


Figure 15. Create a syntactic grammar

Although there is an unlimited number of queries that one can store in a single grammar file, each NooJ grammar contains one, and only one, special rule named **Main**. When applying the grammar to a text or a corpus via the function **Locate**, NooJ will always apply the rule **Main**.

Each query may contain references to other queries: this allows grammar writers to combine elementary queries into more sophisticated ones. In order for NooJ to distinguish names of query from ordinary symbols, one must use the colon character ":" as a prefix to the rule name For instance, consider the grammar **_death.nog** in the English module.

In the following figure, the rule **Main** refers to the four rules **Verbs**, **Nouns**, **Adjectives** and **Expressions**.

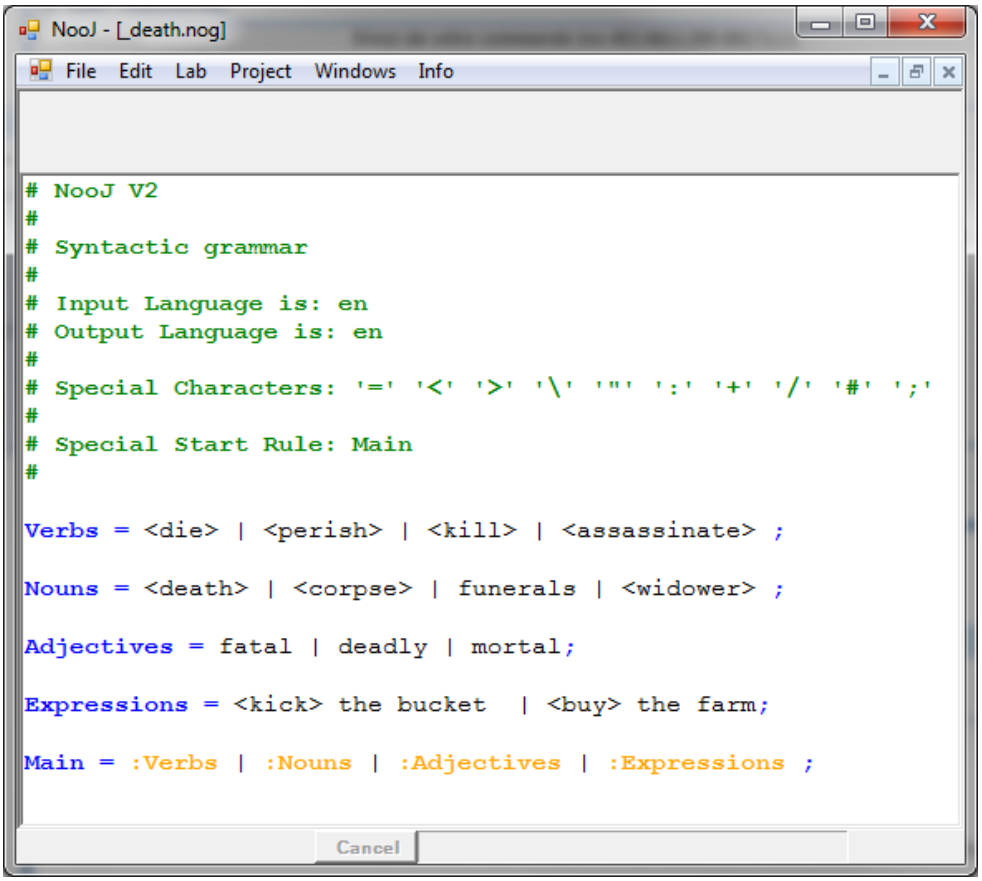


Figure 16. A syntactic grammar

When applying this grammar to a text or a corpus, NooJ actually applies the query named **Main**, which is a combination of the other four queries.

In order to apply this grammar to a text or a corpus, check the option "a NooJ grammar" in the Locate window:

ck option

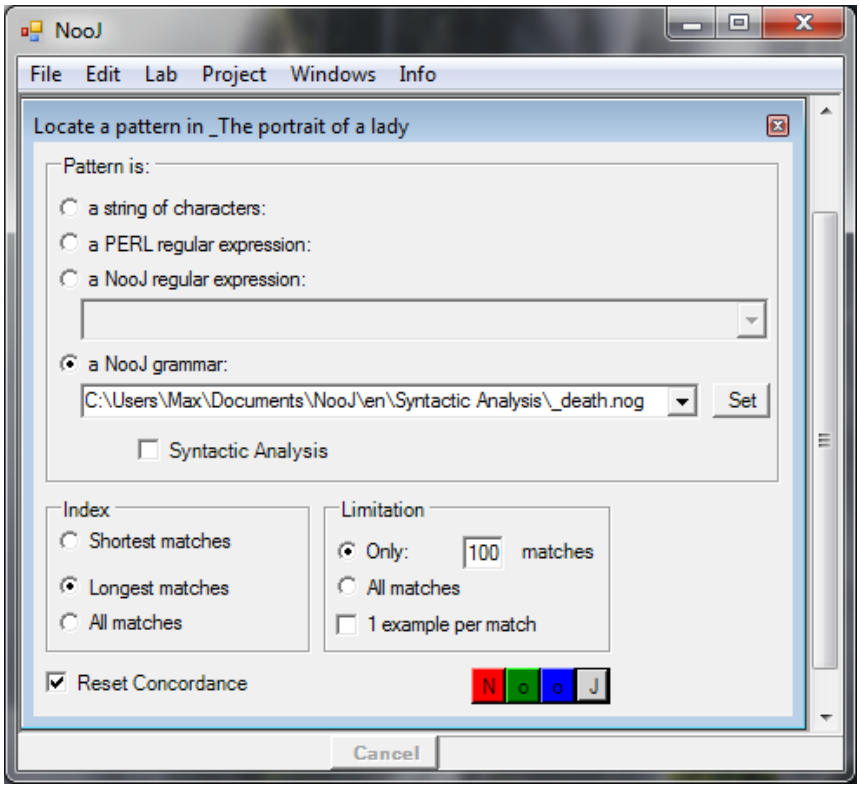


Figure 17. Applying a grammar to a text

Context-Free Grammars

NooJ grammars are more powerful than regular expressions. The ability to use auxiliary queries in a grammar allows linguists to create recursive grammar such as the following one:

```
Main = :Sentence ;
Sentence = :NounPhrase <V> :NounPhrase ;
NounPhrase = <DET> <N> | that :Sentence ;
```

Notice how the query Sentence refers to NounPhrase, which in turn refers to Sentence. Grammars that can contain recursive references are named **Context-Free Grammars** (CFGs).

We will see later that NooJ grammars may contain variables that allow linguists to write even more powerful grammars, equivalent to Turing Machines.

Exercises

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

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...". Then generalize the pattern to recognize negations as well as adverbial insertions.

(2) The wordform "*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 only the verbal form, and one to recognize only the preposition.

Start by studying the unambiguous minimal contexts in which this form is unambiguous, for instance "(I | you | we) like", "(should | will) like", "to like".

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

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 eat", "I won't work tomorrow" and "I'll come back in a few weeks".

6. The Grammar Graphical 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 NooJ graphs.

Create a grammar

In NooJ, grammars of all sorts are represented by organized sets of 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 initial node of the graph, and ends at its terminal node.

Select in the menu **File > New > Grammar**. Selecting the language "en" (for English) both in the INPUT and in the OUTPUT parts of the grammar, make sure the option "graphical editor" is set, then click the button: **Syntax**.

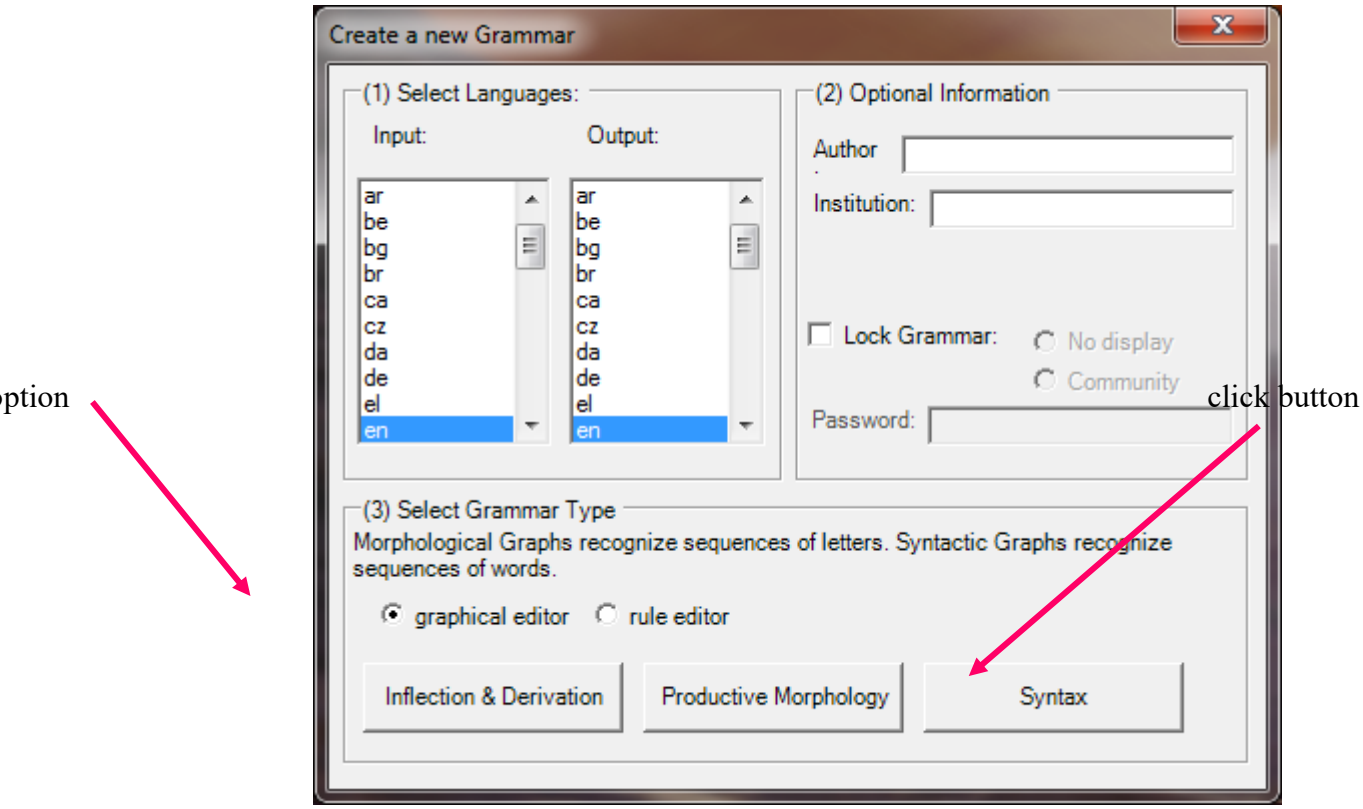


Figure 18. Using the graphical editor

A window like the following figure should be displayed; this grammar contains already two nodes: the initial node is represented by a horizontal arrow, and the terminal node is represented by a crossed 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:

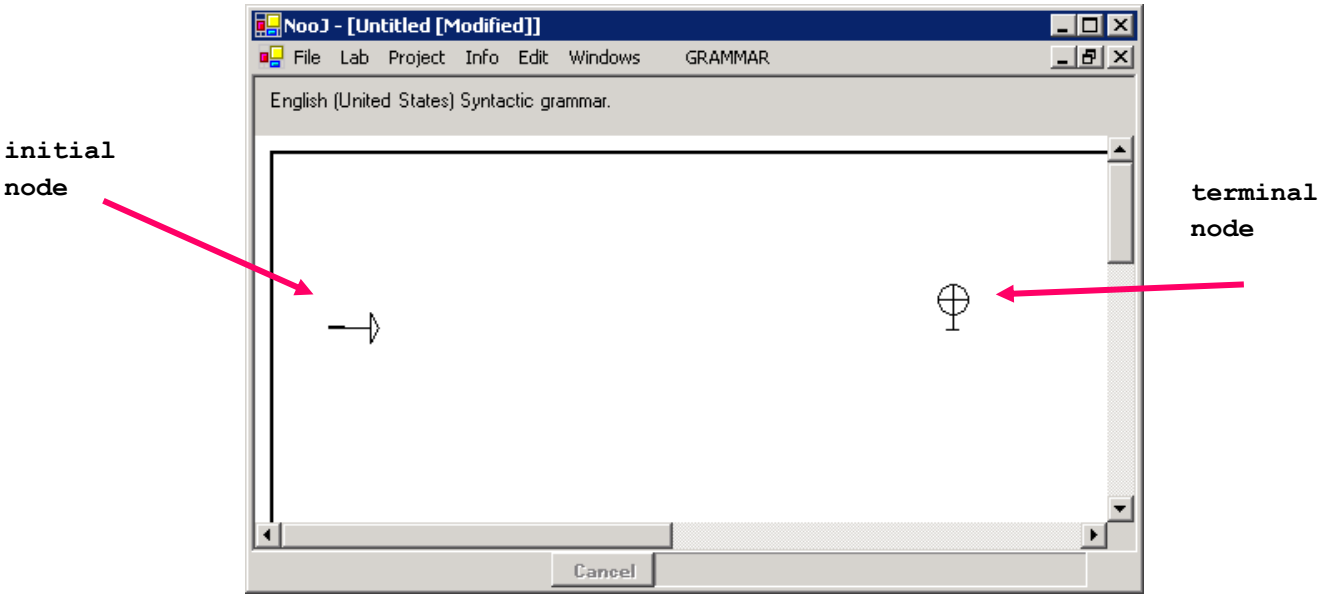


Figure 19. An empty graph contains already an initial node and a terminal node

Basic 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 the text "the" (this will be the label of the node), then validate by hitting **Ctrl-Enter** (press the **Control** key, and then the **Enter** key at the same time).

In order to **select a node**, click it. In order to **unselect a node**, click anywhere on the window (but not on a node). Make sure you deselect the previous node by clicking anywhere in the window (but not on a node). Then, create a node (**Ctrl-Click** somewhere, but not in a node), enter the label "<WF>", then validate with **Ctrl-Enter**. Finally create a third node labeled with "this".

In order to delete a node, select it (click it), then erase its label, then validate with the **Ctrl-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, NooJ 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: connect the initial node to the node labeled "the", and then to the node labeled "this". Then select the two nodes "the" and "this", and connect them to the node "<WF>". Then connect the latter node to the terminal node. You should obtain a graph like the following:

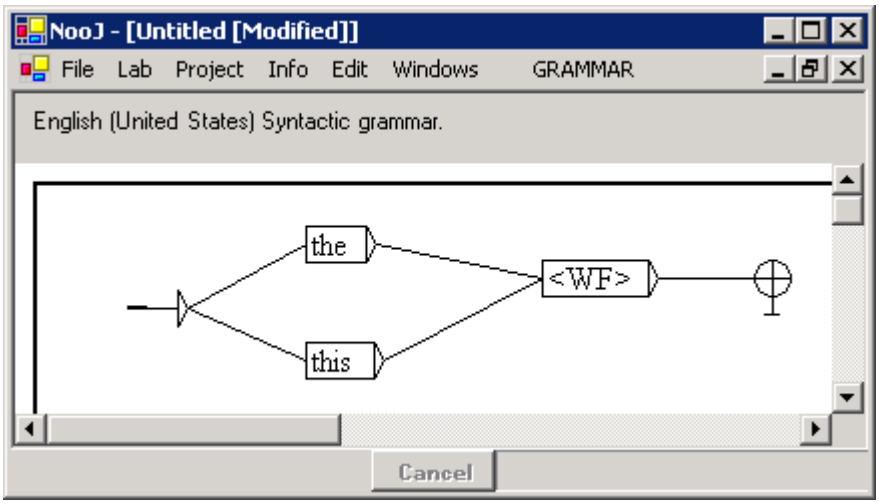


Figure 20. Graph that recognizes "the" or "this", followed by any wordform

This graph recognizes all the sequences that start with "the" or "this", followed by any wordform (remember that "<WF>" stands for any wordform). For instance: "this cat", or "the house". Make sure to save your grammar: **File > Save** (or Ctrl-S); give it a name such as "my first grammar".

You might have made mistakes, such as:

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

Apply a grammar 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", then call the Locate Panel (**Text > Locate Pattern**):

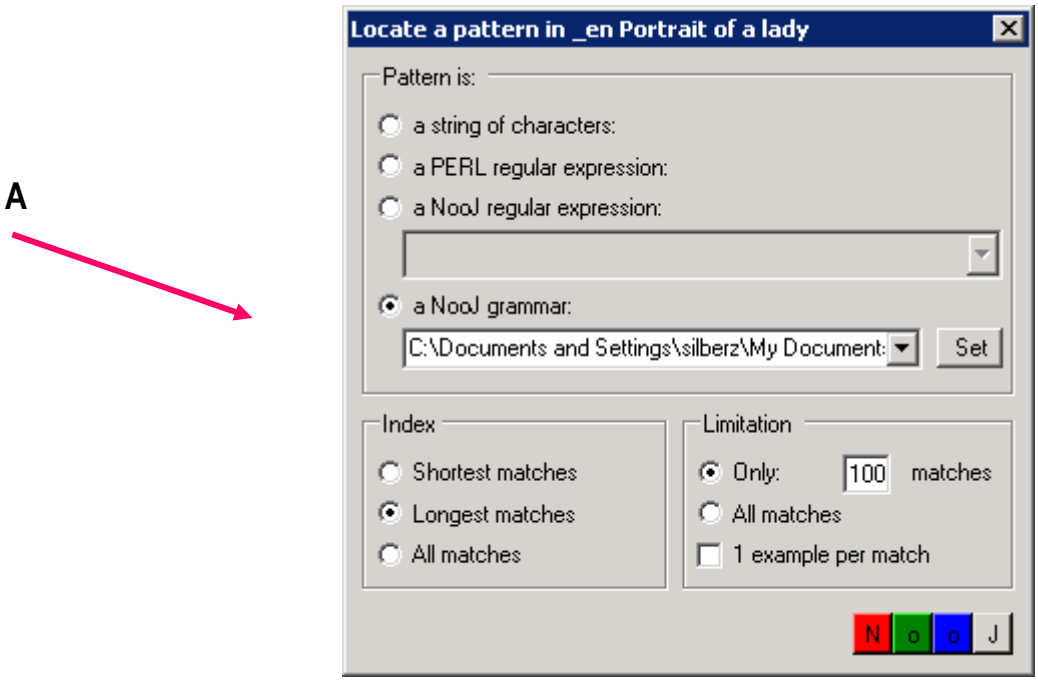


Figure 21. Applying a graph to a text

This time, instead of entering a regular expression, we are going to apply a grammar. **(A)** Select the option **a NooJ grammar**; 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 one colored button at the bottom right end of the "Locate panel". NooJ launches the search, then displays the number of matches found; click **OK**. You should get a concordance similar to the following figure:

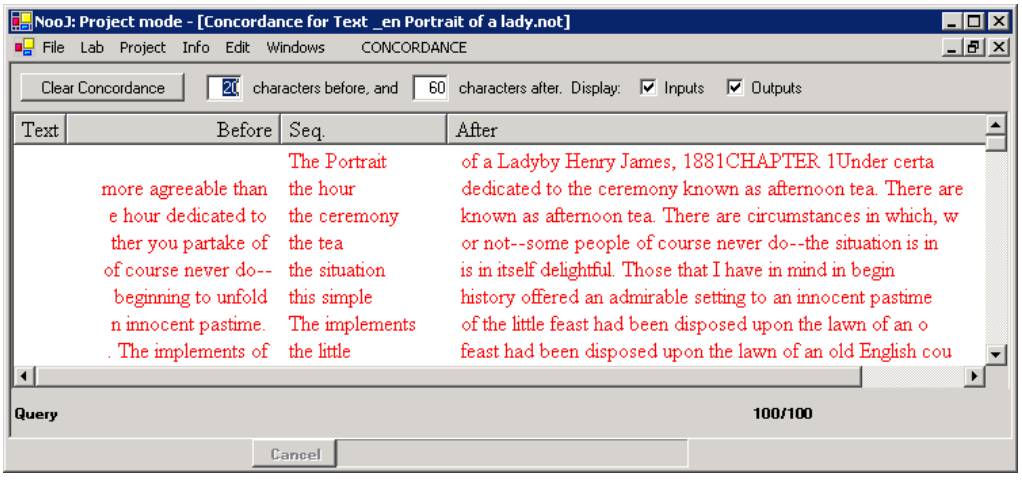


Figure 22. Concordance of the grammar equivalent to the regular expression: *(the+this) <WF>*

Create a second grammar

Select in the menu **File > New > Grammar**, then select the languages English/English, then click the "syntactic grammar" button. A new empty grammar is displayed (that already contains the initial and terminal nodes). Create three nodes with the following labels:

there	is	no
this	are	none
	was	not
	were	nothing

To write a disjunction (e.g. *there* or *this*), press the **Enter** key, so that each term appears on one line. Finally, connect the nodes as in the following figure. Save the graph (**File > Save**) with a name such as "my second grammar".

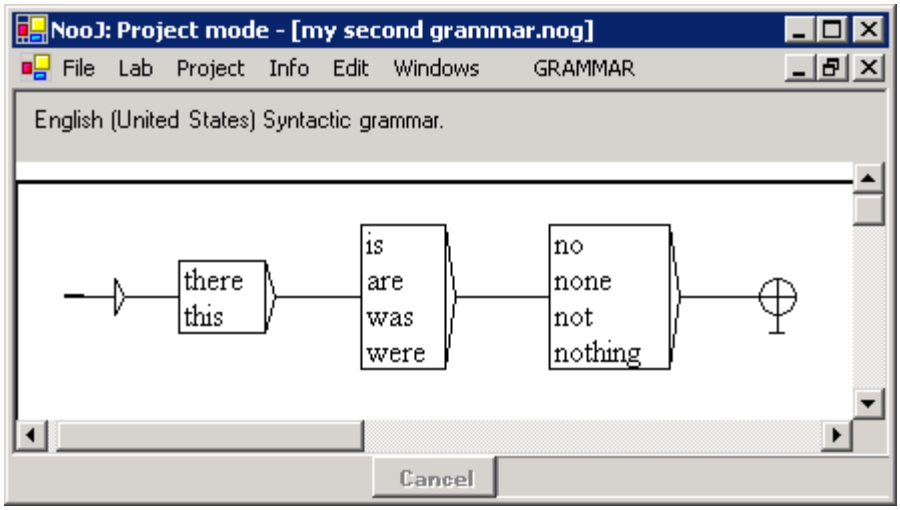


Figure 23. Another grammar

Open the Locate panel (**TEXT > Locate**), select the option **a NooJ grammar**, select your grammar file name, then click a colored button. NooJ applies the grammar to the text, and then displays the concordance.

Note that, because linguistic resources are available for this text, one could have entered the symbol "<be+3>" (any conjugated form of "*to be*", conjugated at the third person), instead of the expression "is+are+was+were".

Describe a simple linguistic phenomena

Grammars 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* (= *he*) and the following verb.

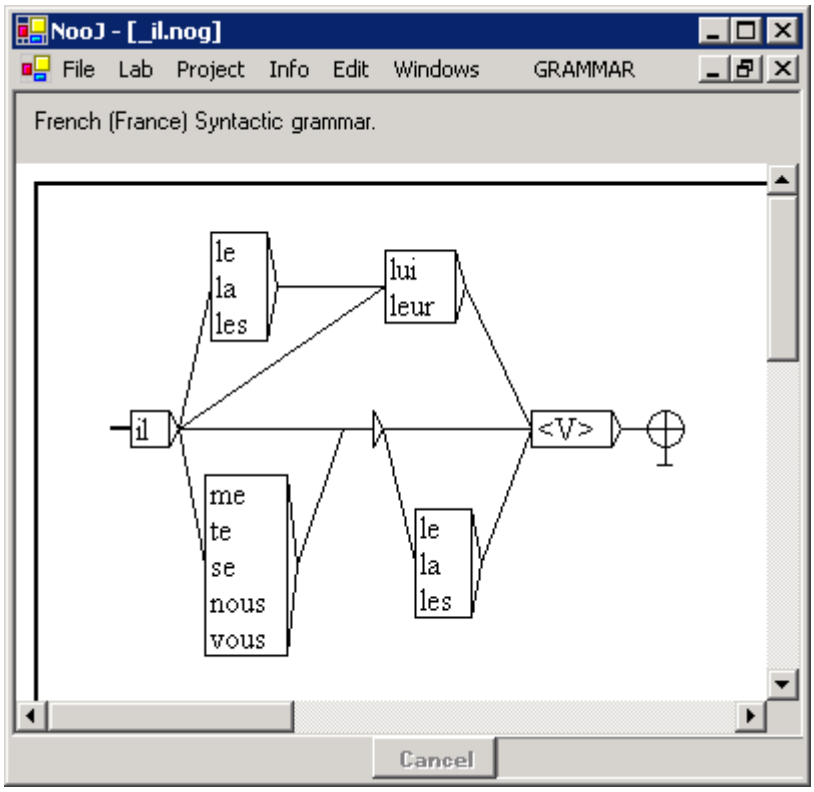


Figure 24. A local grammar for preverbal particles

This graph recognizes the following valid French sequences:

Il dort (he sleeps),
Il le lui donne (he gives it to him),
Il leur parle (he talks to them),
Il me la prend (he takes her from me)

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

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

Exercise: build this grammar, 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, load the text "La femme de trente ans", and apply the grammar to the to study its coverage.

Optimize a grammar

Grammars can contain a large number of embedded graphs. For grammars that are relatively stable and are used often, it might be worth it to optimize them by compiling one "flat" graph, equivalent to the structured grammar. Although the flattening of a grammar might take a long time, and the resulting file is much larger than the original one, the resulting .fst file should run much faster than the original grammar.

Use the command GRAMMAR > Optimize Grammar to compile the grammar into an optimized .fst file; use the file filter ".fst" at the bottom right of the Locate Panel to select a NooJ Grammar stored in a .fst file (as opposed to the default .nog file).

CORPUS PROCESSING

Up until now, we've been working with small text examples, or with the file "_en Portrait of a Lady.not" which is distributed with NooJ. This file, as well as all ".not" files, are text files that are stored in NooJ's format, which means that it contains the text associated with linguistic information: usually, the text has been delimited into text units; each text unit is associated with a Text Annotation Structure; a set of dictionaries, morphological and syntactic grammars have already been applied to the text.

We see in 7 how to import and process "external" text files, either "raw" (such as texts edited from MS-Windows Notepad), documents that were created from word-processing programs such as MS-Word, Web pages that were pulled from the Web, etc. NooJ's .not text files can also be exported as XML documents. In 0, we will discuss the construction and management of Corpora with NooJ. A corpus is a set of text files which share the same characteristics (e.g. language, file formats and structure), that NooJ can process as a whole.

7. Importing & Exporting Text files

NooJ uses its own file format to process texts. Basically, ".not" NooJ text files store the text as well as structural information (e.g. text units), various indices and linguistic annotations in the Text Annotation Structure.

To open a NooJ text file, use the command **File > Open > Text**.

To create a new text file, use the command **File > New Text**. NooJ then creates an empty ".not" file that is ready to be edited, saved and processed. Note that in that case, you will have to perform a Linguistic Analysis (**TEXT > Linguistic Analysis**) before being able to apply queries and grammars to the text.

Any ".not" file can be modified. In order to edit a .not file, click **TEXT > Modify**. Note that as soon as you modify a text file, NooJ erases all its Text Annotation Structure. Therefore, you will need to perform a Linguistic Analysis again before being able to apply queries or grammars to it.

NooJ is usually used to work with "external" text files, i.e. with texts that were constructed with another application. We need to import them in order to create a ".not" file and thus to parse them with NooJ.

In order to import a text file, select **File > Open > Text**, and then, at the bottom of the window, select the option "Files of type: Import Text".

When importing a text file, three parameters need to be set:

The text's language

(See A) Although NooJ's multilingual engine can process texts that contain multilingual contents, it can perform a given linguistic analysis in only one language (each instance of its linguistic engine works in "monolingual mode"). That is to say, any instance of NooJ's linguistic engine applies dictionaries that belong to one, and only one language. When using NooJ's Graphical User Interface, the option "Select Language" is used to set the current language, i.e. the set of linguistic data that will be applied to the text when performing the next "Linguistic Analysis" command.

The text's file format

(See B) NooJ understands all variants of DOS, EBCDIC, ISCII, ISO, OEM, Windows, MAC and UNICODE character encodings, as well as specific file formats such as Arabic ASMO, Japanese EUC and JIS variants, Korean EUC and Johab, etc.

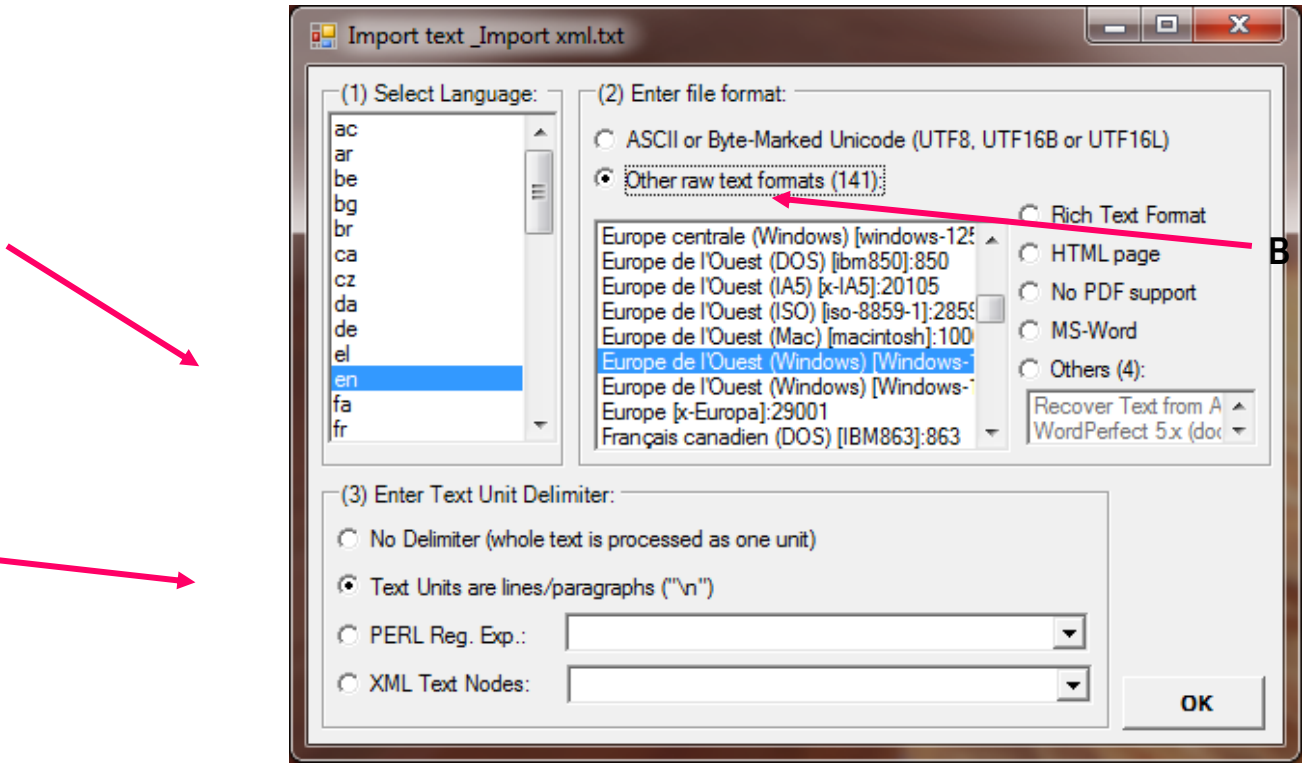


Figure 25. Import a text

On certain Windows 2000 or XP systems, support for some of these file formats might not be already installed on the PC. For instance, during the installation of the French version of Windows XP, support for Asian languages and file formats is not installed by default. If a language or a file format is not supported by your PC, install them by going to the Windows control panel, then to "Regional and Language Options", and then by setting the list of supported languages and file formats.

The file format selected by default for importing texts is "ASCII or Byte-Marked Unicode (UTF8, UTF16B or UTF16L)". That means that characters should be

encoded in ASCII (the "US-only" version of ASCII, i.e. 128 characters without accents), or in one of three variants of Unicode, as marked by a special code inserted at the very beginning of the text file.

If you are trying to import a text file that was created with a Windows application such as notepad, you should select the file format "Other raw text formats": NooJ will select the file format used by your OS by default, e.g. "Western European Windows" on French PCs.

Find out what your text file format is before importing it into NooJ. If you notice that the text you have imported has lost all its accents, chances are that you selected the default "pure ASCII" format, instead of one of its extended version. Close the text file, and then re-import it with its correct file format.

NooJ understands a number of other file formats that are used to represent structured documents, such as RTF, HTML, PDF[5], all variants of MS-WORD (PC, Windows or Mac), as well as a number of "other document file formats" that can be opened by MS-WORD import text functionality, such as WordPerfect, Outlook, Schedule, etc.

In order to process documents represented in any of MS-Word formats, as well as the "other documents" file formats, NooJ actually connects to the MS-WORD application to launch its "text load" functionality. Therefore, importing these documents is only possible if MS-Word is installed on your computer.

Text Unit Delimiters

(See C) When NooJ parses a text, it processes the Text's Units (TUs) one at a time. In consequence, NooJ cannot locate a pattern that would overlap two or more text units. It is important to tell NooJ what the text's units are, because they restrain what exactly NooJ can find in the text.

If one works on Discourse Analysis, to study how sentences are organized together, or tries to solve anaphora that might span over several paragraphs, then Text Units should be as large as possible.

On the other hand, if one studies the vocabulary of a corpus, or even co-occurrences of wordforms in a corpus, then Text Units should be as small as possible.

Enter the corresponding Text Unit Delimiter:

[5] In order for NooJ to process non-encrypted PDF files, one needs to install the free software pdf2t2xt.exe in NooJ's _App folder.

(1) No delimiter:

The whole text is seen as one large text unit. NooJ can find patterns that span over the whole text, e.g. can locate the co-occurrence of the first and the last wordforms of the text.

This option is particularly useful when texts are small, and the information to be extracted is located everywhere in the text. For instance, a semantic analysis of a technical news bit (weather report, medical statement, financial statement) that aims at producing a semantic predicate such as:

```
Admission ( Date (3,10,2001),
            Location (Toronto),
            Patient (Sex (F),Age (50)) )
```

from a text in which each of the elementary piece of information (location, date, etc.) could be located in different paragraphs, from the very beginning of the text file, to its very end.

(2) Text Units are lines/paragraphs:

This is the default option. NooJ processes the character "New Line" (also noted as "\n" by programmers and as "^p" in MS-Word), or the sequence of two characters "New Line / Carriage Return" (also noted as "\n\r"), as a text unit delimiter.

The "New Line" character is used either as a line break, for instance in poems, or more generally as a paragraph break, by word processors including Windows' Notepad tool and MS-Word. In the latter case, NooJ will process the text paragraph per paragraph.

(3) PERL regular expression:

This option allows users to define their own text unit delimiter. In the simplest cases, the delimiter could be a constant string, such as "===" in the following example:

```
This is the
first text unit.
===
This is the second text unit.
===
This is
the third
text unit.
===
This is the fourth text unit.
```

PERL regular expressions allow users to describe more sophisticated patterns, such as the following:

```
^[0-9][0-9]:[0-9][0-9][ap]m$
```

This expression recognizes any line that consists of two two-digit numbers separated by a colon, followed by an "a" or a "p", followed by "m", as in the following text:

```
12:34am
This is the first text unit.
07:00pm
This is the second text unit.
```

PERL regular expressions can contain special characters such as "^" (beginning of line), "$" (end of line), "|" (or), etc. Look at PERL's documentation for more information.

Importing XML documents

The last option allows users to process structured XML documents, more specifically texts that contain XML-type tags. Here is an example of such texts:

```
<document>
<page>
<s>this is a sentence</s></page>
<page><s>this is another sentence</s>
<s>the last sentence</s>
</page></document>
```

This text is structured as one **document** block of data, inside which there are two **page** blocks of data. The first page contains one **s** ("sentence"); the second page contains two sentences. Note that each level of the document structure is spelled as some data between by two tags: one beginning tag: **<document>**, **<page>** and **<s>**, and the corresponding ending tag: **</document>**, **</page>** and **</s>**.

Text Nodes

When parsing such structured documents with NooJ, it is important to tell NooJ where to apply its linguistic data and queries, i.e. what are the nodes in the structured data that include the textual data to be processed.

Typically, a document may contain meta-data such as the author name, the date of publication, and references to other documents, as well as textual information such as an abstract, a text and a conclusion:

```
<document>
<author>Max Silberztein </author>
<date>August 21, 2006</date>
<abstract>this is an abstract</abstract>
<text>this is a very short introduction</text>
<conclusion>this is the conclusion</conclusion>
</document>
```

In that case, we would ask NooJ to apply linguistic data and queries to the blocks of textual data such as **<abstract>**, **<text>** and **<conclusion>**, and simply ignore the other data. In order to specify what blocks of textual data we want NooJ to process, we give NooJ the list of all corresponding tags, e.g.:

```
<abstract> <text> <conclusion>
```

Multilingual texts

NooJ uses this mechanism to process multilingual texts: for instance, a multilingual text might look like:

```
<document>
<text-en>this is an English sentence</text-en>
<text-fr>ceci est une phrase française</text-fr>
<text-en>another sentence</text-en>
<text-fr>une autre phrase</text-fr>
...
</document>
```

In that case, we can open the text as a French "fr" text and then select the corresponding **<text-fr>** text node to parse, and/or open the text as an English "en" text, and then select the **<text-en>** blocks of text.

Attribute-Value pairs

XML tags can be associated with attribute-value pairs that NooJ can process. For instance, instead of having two different XML tags <text-fr> and <text-en>, we might use one single XML tag, say <text> which has two possible values, lang=fr or lang=en. The corresponding document would look like:

```
<document>
<text lang=en>this is an English sentence</text>
<text lang=fr>ceci est une phrase française</text>
<text lang=en>another sentence</text>
<text lang=fr>une autre phrase</text>
...
```

</document>

In that case, we would ask NooJ to select the text nodes **<text lang=en>** or **<text lang=fr>** for linguistic analysis.

Importing XML information into NooJ's Text Annotation Structure

When NooJ imports an XML document, it automatically converts XML tags to NooJ annotations.

By default, XML tags are converted into syntactic/semantic annotations. These annotations are the ones displayed in green in the Text Annotation Structure. The head of the XML tag is converted into a NooJ category, each XML attribute-value pair is converted into a NooJ name-value property field, and each XML attribute is converted into a NooJ feature. For instance, the following XML text:

```
<DATE>Monday, June 1st</DATE>
```

will produce NooJ's annotation **<DATE>**, associated to the text "Monday, June 1st". In the same manner, the XML text:

```
<NP Hum Nb="plural">Three cute children</NP>
```

will produce NooJ's annotation **<NP+Hum+Nb=plural>**.

All imported XML tags are translated into syntactic/semantic annotations, except the special XML tag **<LU>** (LU stands for "Linguistic Unit"). NooJ understands LU's as Atomic Linguistic Units; LUs require a category property, and may have a lemma property. For instance, the following XML text:

```
<LU CAT=DET plural>The</LU>
<LU LEMMA="child" CAT=N plural>children</LU>
```

will be converted into the annotations: **<the,DET+plural>** and **<child,N+plural>**.

Notice that the LEMMA is optional: if absent, the lemma is identical to the wordform.

The ability to import XML tags as NooJ's lexical or syntactic/semantic annotations allows NooJ to parse texts that have been processed with other applications, including taggers.

Exporting Annotated Texts into XML documents

NooJ can also export a Text Annotation Structure as an XML document. To do so, use the TEXT > Export annotated text As an XML document.

NooJ's annotations will be represented as XML tags and inserted in the resulting text. There are five important cases to consider:

(1) Syntactic annotations are converted directly as XML tags. For instance:

```
<DATE>Monday, March 13th</DATE>
```

Syntactic annotations may have properties. In that case these properties are converted into XML tag's properties. For instance:

```
<PERSONNAME GENDER="masc" DOMAIN="politics">Bruno
Smith, head of the U.N. Organization</PERSONNAME>
```

Syntactic annotations may have embedded syntactic annotations. For example:

```
<DATE>Monday, March 13th <HOUR>at seven in the
morning</HOUR></DATE>
```

(2) Lexical annotations are represented as **<LU>** XML tags in the resulting text. The annotation's lexical properties are converted into XML tag's properties. For instance:

```
<LU LEMMA="eat" CAT=V TENSE=PR PERSON=2
NUMBER=s>eats</LU>
```

(3) The special syntactic annotation **<TRANS>** is used to perform translations or replacements in the text. For instance, **<TRANS+EN>** will replace all the text that was annotated with the **<TRANS>** annotation with the value of their property "**EN**". For instance, consider the following French text:

```
C'est arrivé lundi 2 mars.
```

If in this text the sequence "lundi 2 mars" has been annotated with the following annotation:

```
TRANS+EN="Monday, March 2nd"
```

then in the resulting XML text, it will be replaced with the following text:

```
C'est arrivé <EN>Monday, March 2nd</EN>.
```

This option allows users to produce local translations that can be accumulated by cascading local translation grammars: one grammar could recognize and translate date complements, another grammar can recognize and translate named entities, another some specific noun phrases, etc.

(4) The option **Export Annotated Text Only** will only export sequences of the text that are annotated with the specified annotations, and simply ignore the remaining text.

(5) In case of ambiguity:

-- NooJ produces annotations by scanning the text from its beginning to its end (from left to right for most languages; from right to left for Semitic languages). Therefore, in case of multiple matches, it will always give priority to the first annotation encountered. For instance, consider the following text:

```
The red army badge is on the shelf.
```

If NooJ has annotated both terms "red army" and "army badge" with the same annotation <TERM>, exporting this text as an XML document will produce the following result:

```
The <TERM>red army</TERM> badge is on the shelf.
```

The term *army badge* will then be ignored in the resulting XML document, even though it is represented in NooJ's TAS. Note that this limitation means that users should always export a NooJ TAS at the very end of the analyses process, since each "Export to XML" process might loose some information. Is is a bad idea to include multiple NooJ analyses between import and exports of XML texts, such as in the following flow:

Import an XML document => annotate dates with NooJ => export the result as an XML document => reimport the XML document in NooJ => annotate named entities with NooJ => export the result as an XML document => reimport the result in NooJ => annotate noun phrases => export the result

whereas the following flow is much better:

Import an XML document => annotate dates with NooJ, then annotate named entities with NooJ, then annotate noun phrases => export the result

-- if a sequence in the text is annotated with two or more different annotations with the same length, or with different lengths, NooJ will export all of them. For instance:

```
<POLITICS><PERSONNAME>Bruno Smith</PERSONNAME>, head of
the U.N. Organization<POLITICS>
```

Options:

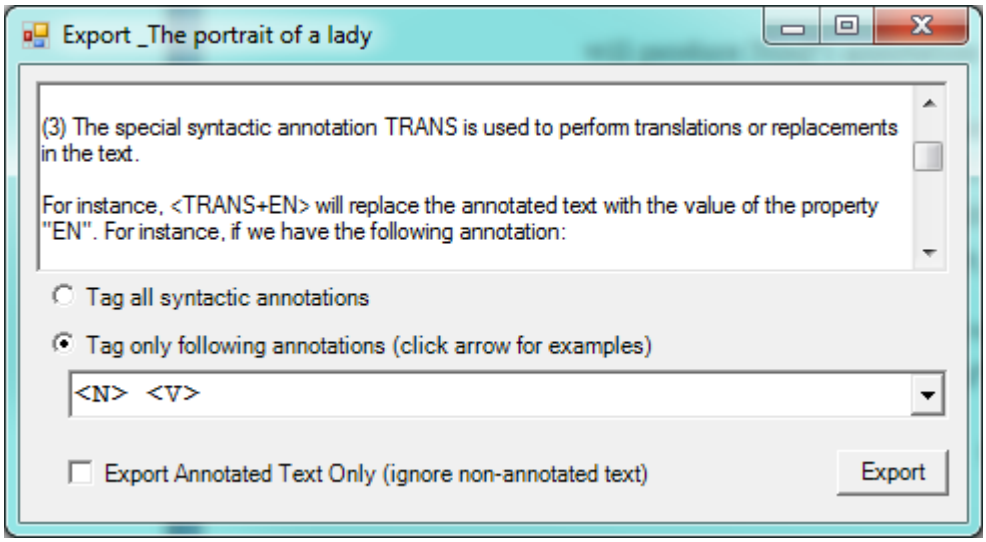


Figure 26. Export a TAS as an XML document

-- Tag all syntactic annotations:

Every sequence in the text that was annotated by a syntactic annotation (an annotation displayed in green in the TAS) will be written inside a pair of XML opening/ending tag. For instance:

```
It happened <DATE>Monday, March 13th</DATE>.
```

-- Tag only following annotations:

This option allows the user to specify the annotations that will be written between XML tags in the resulting text. Note that this option allows the user to select lexical, morphological or syntactic annotations. For instance, if the user specifies the annotations <N> and <DATE>, then the output will look like:

```
The <N Number="s">table</N> <V Lemma="be" Tense="Pr"
Person="3" Number="s">is</V> in the <N Number="s">room</N>.
```

exported Every sequence in the text that was annotated by a syntactic annotation (an annotation displayed in green in the TAS) will be written inside a pair of XML opening/ending tag. For instance:

8. Working with corpora

A corpus is a set of text files that share the same parameters: usually, the language, the structure and the encoding.

NooJ uses its own file format to process corpora. Basically, ".noc" NooJ corpora files store the texts as well as their structural information (e.g. text units), various indices and linguistic annotations in each Text Annotation Structure.

To open a NooJ corpus file, use the command **File > Open > Corpus**. To create a new corpus file, use the command **File > New Corpus**. NooJ then creates an empty ".noc" file in which one can import sets of text files.

When creating a new corpus file, the same three parameters that were used to import texts have to be set when creating a corpus file: (A) the Corpus' language, (B) the corpus' files' format, and (3) the texts' delimiters.

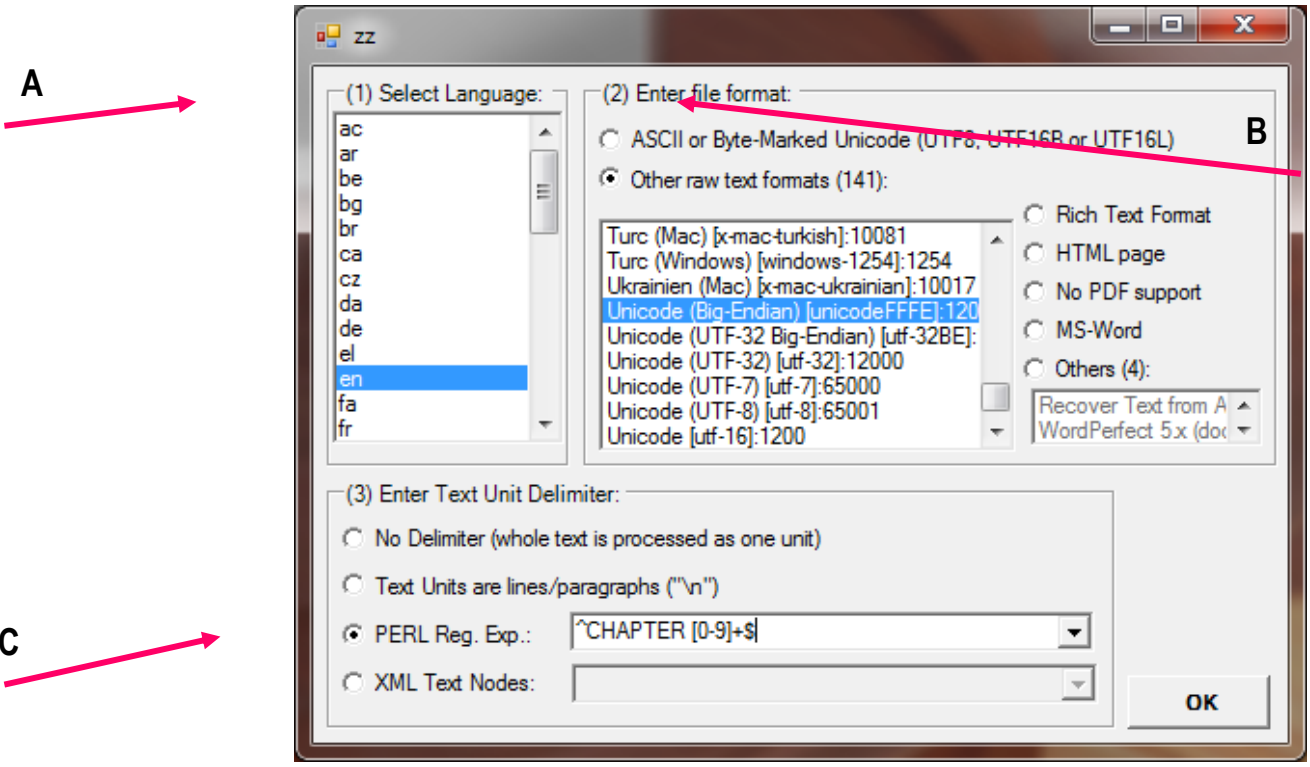


Figure 27. Create a corpus

All NooJ's functionalities that can be performed on one text can also be performed at one corpus' level. It is then possible to compute statistical measures at the characters, tokens, digrams, annotations and unknown levels; it is possible to perform the linguistic analysis (**CORPUS > Linguistic Analysis**) at the corpus level, as well as to build concordances at the corpus level.

Note that concordances built from a corpus have an extra column that displays the text's name from which each occurrence was extracted; moreover, the statistical report (**CONCORDANCE > Build Statistical Report**) now computes the standard deviation of the number of occurrences for each text file.

LEXICAL ANALYSIS

The first level of text analysis requires the computer to identify the **Atomic Linguistic Units** (ALUs) of the text, i.e. its smallest non-analyzable elements. In chapter 9, we define these ALUs, and we show how NooJ's dictionaries are used to describe them. In Chapter 10, we describe NooJ's inflectional and derivational morphological module; in chap. 11, we present NooJ lexical and morphological grammars. In Chapter 12, we present NooJ's lexical parser, which uses dictionaries, the inflectional and derivational engine, and morphological grammars to annotate words in texts.

9. NooJ Dictionaries

Atomic Linguistic Units (ALUs)

Atomic Linguistic Units (ALUs) are the smallest elements that make up the sentence, i.e. the non-analyzable units of the language. They are "words" which meaning cannot be computed nor predicted: one must learn them to be able to use them. For instance, one need to learn how to say "chair" or "French fries" in another language: there is no linguistic property for these words that could be used to predict or compute their correct translations. Moreover, even for the purpose of analyzing only English texts, one has to describe explicitly all the syntactic and semantic properties of these words, e.g. **Concrete**, **Vegetable**, **Food**, etc. because none of these properties can be deducted from their spelling or their components. Therefore, it is crucial to describe these ALUs explicitly.

It is the role of NooJ's **lexical analysis module** to represent ALUs. From a formal point of view, NooJ separates the ALUs into four classes:

Simple Words are ALUs that are spelled as wordforms. For example, apple, table.

Affixes are ALUs that are spelled as sequences of letters, usually inside wordforms. For example: dis- and -ization in the wordform "disorganization".

Multi-word Units are ALUs that are spelled as sequences of wordforms. For example: round table (when that means "meeting"), nuclear plant (when that means "Electricity producing industrial plant").

Frozen Expressions are ALUs that are spelled as potentially discontinuous sequences of wordforms. For example: take ... into account.

The terminology used here is natural for English and Romance languages such as French, Italian or Spanish; in the Natural Language Processing community, researchers

use both terms "compound words" and "multi-word units" indistinctly. However, the term "compound word" is less well suited for Germanic languages, where we are accustomed to using the term "compound word" to refer to an *analyzable* sequence of letters (In NooJ, **multi-word units** or **compound words** are sequences of *non-analyzable* sequences of wordforms). This terminology is irrelevant for Asian languages such as Chinese, where there are no formal differences between affixes, simple words and multi-word units, because there is no blank between wordforms.

What is important to consider, however, is that no matter what the language, ALUs can always be grouped into four classes or less, which each correspond to an automatic recognition program within NooJ's lexical parser: in order to identify the first and third types of ALUs (simple words and multi word units), NooJ will look up its dictionaries. The second type of ALUs ("affixes") will be identified by breaking wordforms into smaller units. NooJ will recognize the fourth type by applying syntactic grammars.

Caution: do not confuse the terms simple word (which is a type of ALU) and wordform (which is a type of token). A *wordform* is a sequence of letters that does not necessarily correspond to an Atomic Linguistic Unit: for example, "*excthz*" is a wordform, but not an English word; "*redismountability*" is a wordform, but not an ALU because it can be reduced into a sequence of ALUs: "re", "dis", "mount", "able", "ility". A *simple word* is, by definition, an ALU, i.e. the smallest non-analyzable element of the vocabulary.

The **recognition** of an ALU by the NooJ's lexical parser does not imply that the ALU truly occurs in the text. For example, the fact that we find the sequence of two wordforms: "round table", in NooJ's dictionary does not necessarily mean that the multi-word unit meaning "a meeting" 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 retreat would be considered ambiguous (*Napoleon's retreat* vs. *John retreats the problem*).

In NooJ, click **Info > Preferences**, select the English language "en", then click the tab **Lexical Analysis**. The page that is displayed contains two areas: **Dictionary** and **Morphology**. The **Dictionary** zone contains all the lexical resources that are used to recognize simple words and multi-word units; the morphology zone displays all the morphological grammars that are used to recognize wordforms from their components (prefixes, affixes and suffixes).

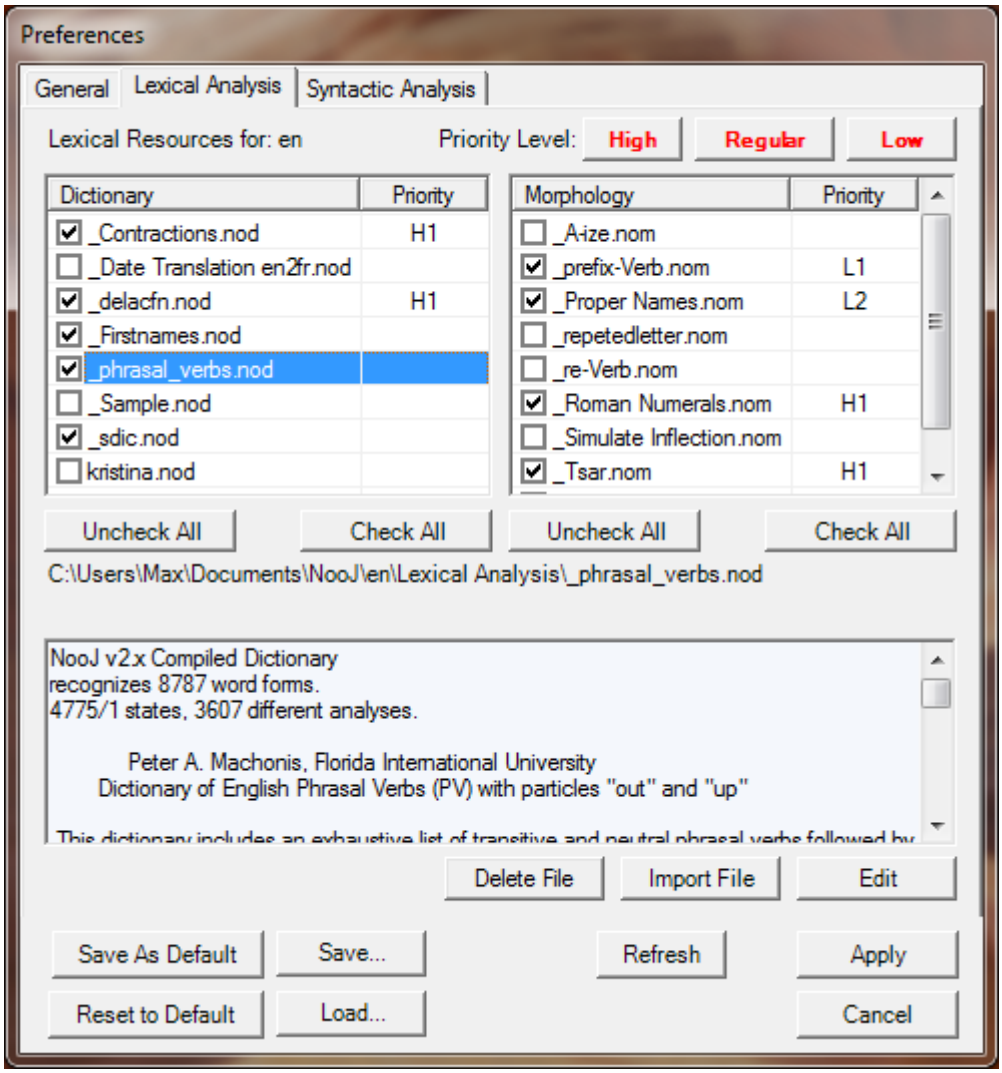


Figure 28. Resources used to recognize atomic linguistic units

NooJ dictionaries are used to represent, describe and recognize simple words, multi-word units as well as discontinuous expressions. The dictionaries displayed here are ".nod" files that have been compiled from editable ".dic" source files.

Technically, ".nod" files are finite-state transducers; we speak of them as compiled dictionaries because, from the point of view of the end user, they come out of editable text ".dic" type documents that were compiled using the **Lab > Dictionary > Compile** command.

NooJ offers two equivalent tools to represent and describe inflectional and derivational morphology:

> **Inflectional / derivational descriptions** are organized sets of Context-Free rules that describe morphological paradigms.

Inflectional / Derivational grammars are structured sets of graphs that describe morphological paradigms.

Both sets of rules are stored in NooJ inflectional/derivational grammars, in ".nof" files. These descriptions are lexicalized, i.e. each lexical entry of a NooJ dictionary must be associated with one or more inflectional and derivational paradigms, and inflectional or derivational paradigms do not work if they are not associated with lexical entries.

Finally, NooJ offers also **morphological grammars** (".nom" files) to recognize certain wordforms, for which an inflection or a derivational description is not adequate. For instance, in Arabic or in Hebrew, the preposition and the determiner are concatenated to the noun: we don't want to describe these complex wordforms as inflected variants of the noun, but rather as a sequence three ALUs. In the same manner, Germanic languages have free noun phrases that are spelled as wordforms; NooJ needs to segment these noun phrases into sequences of ALUs. Even in English, wordforms such as "cannot" need to be parsed as sequences of ALUs (i.e. "can" "not").

Attention INTEX users: NooJ dictionaries, as opposed as INTEX's, contain the full description of the inflection and the derivation of their entries. This explains why NooJ does not need DELAF-type dictionaries anymore. More importantly, whether INTEX could only lemmatize wordforms (e.g. "was" → "be"), NooJ can generate any derived or inflected form of any wordform (e.g. "eats" → "eaten")

Finally, we will see that frozen expressions (the fourth type of ALUs), as well as semi-frozen expressions, are described in NooJ's Syntactic component: click **Info > Preferences**, select the English language "en", then click the tab **Syntactic Analysis**. The page that is displayed contains a list of syntactic grammars. We will see later how to formalize frozen expressions.

Dictionaries' format

Attention INTEX users: NooJ dictionaries are a unified version of DELAS, DELAC and DELAE dictionaries. NooJ does not need DELAF nor DELACF dictionaries. NooJ dictionaries contain indistinctly both simple words and compound words (aka *multi-word units*), and can represent spelling or terminological variants of lexical entries.

Generally, the dictionary of a given language contains all of the lemmas of the language, and associates them with a morpho-syntactical code, possible syntactic and semantic codes, and inflectional and derivational paradigms. Here, for example, are some entries from the English dictionary:

```
a,DET+s
aback,ADV
abacus,N+Conc+FLX=CACTUS
abandon,N+Abst+FLX=APPLE
abandon,V+tr+FLX=ASK
abandoned,A
a lot of,DET+p
...
```

The first line represents the fact that the word "a" is a determiner (**DET**), in the singular form (**+s**). The second line describes the word "aback" as an adverb (**ADV**). Note that the word "abandon" functions either as a noun (**N**) or as a verb (**V**), therefore it is duplicated: NooJ dictionaries cannot contain lexical entries with more than one syntactic category. In consequence, NooJ's lexical parser will process the wordform "abandon" as ambiguous.

Lexical ambiguity: When a word is associated with different sets of properties, i.e. different syntactic or distributional information, we must duplicate the word in the dictionary. The corresponding wordform will be processed as ambiguous.

In NooJ dictionaries, all linguistic information, such as syntactic features such as "+tr" (transitive) and semantic features such as "+Conc" or "+Abst" must be prefixed with the character "+".

NooJ is case sensitive; for instance the two codes "+s" (e.g. singular) and "+S" (e.g. Subjunctive) are different.

Lexical entries can be associated with features, and also with properties. A lexical property has a name and a value; it is written in the form +name=value; in the example above, we see that the noun "abandon" is associated with the property "+FLX=APPLE". That means that its inflectional paradigm is "APPLE" (i.e. it inflects like "apple"). In the same way, the inflectional paradigm of the verb "to abandon" is "ASK", etc. Inflectional and derivational paradigms are described in **Inflectional/Derivational description files** (see later).

Remember that NooJ does not know what these codes mean, and it cannot verify that the codes used in a query or in a grammar, as entered by the user, are correct. For example, nothing prohibits the user from writing the following query:

```
<ADVERB>* <ADJ> <NOUN>
```

although this query will probably not identify anything, because no NooJ dictionaries use these codes (that is, until someone does!). The advantage of this "freedom of expression" is that users are free to invent and add any code in their dictionary.

Before using lexical symbols in queries or grammars, make sure that they are actually present in the dictionaries you are using!

Now consider the wordform "abandoned". In the dictionary, we see that there is an entry "abandoned" (Adjective). At the same time, we will see later that the inflectional paradigm "ASK", which is the inflectional paradigm for the verb "abandon", produces the conjugated form "abandoned", with two analysis: "Preterit", or "Past Participle". In consequence, NooJ's lexical parser will process "abandoned" as three-time ambiguous.

Morphological ambiguity: when a wordform is associated with more than one morphological analysis, it is processed as ambiguous.

Other, more specialized dictionaries are also available on the NooJ websites http://www.nooj4nlp.net. If you have built such dictionaries and you think that they might be useful for other NooJ uses, please send them to us (with some documentation!).

Lexical variants

NooJ lexical entries can be linked to a "super-lemma", that acts as a canonical form for the lexical entries as well as all their inflected and derived forms:

```
tsar,N+Hum+FLX=APPLE
csar,tsar,N+Hum+FLX=APPLE
czar,tsar,N+Hum+FLX=APPLE
tzar,tsar,N+Hum+FLX=APPLE
...
```

Lexical entries linked to the same super-lemma will be considered as equivalent from NooJ's point of view. Although each lexical entry has its own inflectional paradigm (e.g. "czar" inflects as "czars" in the plural), all the inflected wordforms:

```
tsar, tsars, csar, csars, czar, czars, tzar, tzars
```

will be stored in the same equivalence class, and any of the symbols <czar>, <tzars>, etc. will match the previous eight wordforms.

Lexical variants can be simple or multi-word terms, e.g.:

```
IBM,ibm,N+Company
Big Blue,ibm,N+Company
IBM Corp.,ibm,N+Company
IBM Corporation,ibm,N+Company
Armonk Headquarters,ibm,N+Company
```

```
...
```

In the latter case, the symbol <ibm> will match any of the term's variants.

Super-lemmas, as well as lemmas, must never be spelled in uppercase. NooJ needs to distinguish between queries on syntactic categories, such as "<ADV>" and queries on lemmas (or super-lemmas), such as "<be>".

Inversely, dictionaries can also contain entries which are in fact simple words, for example:

```
board,school board,N+Organization
board,bulletin board,N+Conc
board,circuit board,N+Conc # electronics
```

This allows us to represent ambiguities associated with certain simple words; therefore, the simple word "*board*" will be treated as ambiguous after consultation of the dictionary.

Finally, the super-lemma associated to each entry of the dictionary can also be used in translation applications; for example:

```
carte bleue,credit card,N+Conc
carte routière,road map,N+Conc
carte-mère,motherboard,N+Conc
carte postale,postcard,N+Conc
```

Complex tokenization

Some times, one single token corresponds to more than one Atomic Linguistic Unit. For instance, the wordform "cannot" must be analyzed as a sequence of the two ALUs: "can" (Verb) "not" (Adverb). In a NooJ dictionary, one would represent this tokenization by the following complex lexical entry:

```
cannot,<can,V+PR><not,ADV>
```

where <can,V+PR> and <not,ADV> are lexemes. These lexemes allow the former token to be analyzed as the sequence of the verb "can" (rather than the noun "a can"), followed by the adverb "not".

Similarly, when a sequence of tokens corresponds to more than one ALU, associate the corresponding NooJ dictionary entry with a sequence of lexical constraints:

```
doesn't,<does,do,V+PR+3+s><not,ADV>
```

The lexeme <does,do,V+PR+3+s> has three fields: the wordform "does", the lemma "do" and the corresponding linguistic information, whereas the former lexeme <can,V+PR> had only two because the wordform "can" is identical to its lemma. Finally, most of these contracted sequences are unambiguous; in that case, one should add the +UNAMB feature to these entries, e.g.:

```
I'm,<I,PRO+1+s><am,be,V+PR+1+s>+UNAMB
```

Special Information Codes

Although NooJ allows users to use and invent any code to describe lexical entries, several codes have a special meaning for NooJ:

+NW	non-word
+UNAMB	unambiguous word
+FLX	inflectional paradigm
+DRV	derivational paradigm

The code **+NW** (non word) is used to describe abstract lexical entries that do not occur in texts, and should not be analyzed as real wordforms. This feature is particularly useful when building a dictionary in which entries are stems or roots of words.

The code **+UNAMB** (unambiguous word) tells NooJ to stop analyzing the wordform with other lexical resources or morphological parser. For instance, consider the following lexical entries:

```
round,A
round table,N+UNAMB+Abst
table,N+Conc
```

If this dictionary is used to parse the text sequence "... round table ...", then only the lexical hypothesis "round table = N+Abst" will be used, and the other solution, i.e. "round,A table,N+Conc" will be discarded. In the same way, if the dictionary contains a lexical entry such as:

```
trader,N+Hum+UNAMB
```

then the property **+UNAMB** inhibits all other analyses, such as a morphological analysis of trader = Verb "*to trade*" + nominalization suffix "*-er*".

The properties **+FLX** and **+DRV** are used to describe the inflectional and derivational paradigms of the lexical entry, see below.

Free sequences of words vs. multi-word units

An important problem faced by the project of describing natural languages on a wide scale is that of the limit between multi-word units (that must be described in dictionaries) and free word sequences (that must be described by syntactic grammars). It is evident for those who perform automatic analyses on natural language texts, that in the following examples:

a red neck, a red nail, a large neck

The first Adjective-Noun sequence must be lexicalized: if *red neck* is not included in a dictionary, a computer would be incapable to predict its lexical properties (for example, "**+Hum**" for human noun), and certain applications, such as automatic translation, would produce incorrect results (such as "*cou rouge*" instead of "*péquenaud*" in French).

These two examples are straightforward; but between these two extremes (an idiomatic multi-word unit *vs.* a free series of simple words), there are tens of thousands of more difficult cases, such as:

rock group, free election, control panel

I have adopted a set of criteria that establish limits, within the NooJ framework, between those sequences that should be lexicalized (i.e. multi-word units), and those that we choose not to lexicalize (cf. [Silberztein 1993]). The primary criteria are:

Semantic atomicity: if the exact meaning of a sequence of words cannot be computed from the meaning of the components, the sequence must be lexicalized; it is therefore treated as a multi-word unit.

For example, the noun *group* is used in a dozen or so meanings: *age group, control group, rock group, splinter group, umbrella group, tour group*, etc. Although the word "group" in each of these sequences always refers to a "group of people", each meaning of "group" is different: an *age group* is a subset of a population characterized by their age; it is not an organization. A *control group* is a subset of people used for comparison with others as part of a medical experiment. A *rock group* is a music band. A *splinter group* is an organized set of people who leave a larger organization to form a new one. A *tour group* is a group of tourists who travel together in order to visit places of interest. An *umbrella group* is an organization that represents the interests of smaller local organizations.

Only the first meaning of *group* (i.e. "band") is relevant in the noun phrase *a rock group*, which cannot mean: a *set of rocks*, a *rocker organization*, the *audience at a rock concert*, etc. In the same way, a *tour group* is not a band on a tour; a *control group* is not an organization who aims at controlling something or a music band, etc.

Similarly, the noun "election" is used in a dozen or so meanings:

presidential election, party election, runoff election, general election, free election

Here too, each of these phrases has a different semantic analysis:

presidential election = people elect someone president
party election = a party elects its leader/representative
free election = an election is free = people are free to participate in the election

and, at the same time, inhibits other possible analyses: in a presidential election, presidents do not elect their representative; in a general election, people do not vote for a general, etc.

Is it then possible that the modifier is responsible for the selection of each of the dozen meanings of "election" or "group"? In other words, maybe the simple word "control" could be linked to the concept "experimental test" in such a way that "control", combined with "group", would produce the meaning "test group used for experiments"? Unfortunately, even a superficial study proves this hypothesis wrong. For instance, consider the following noun phrases built with "control":

A control cell, a control freak, a control panel, a control surface, a control tower

A "control cell" is indeed a test cell used after a biological experiment for comparisons with other cells, but a "control freak" is not a "freak" compared with others after some experiments. A "control panel" is a board used to operate a machine. A "control surface" is a part of an aircraft used to rotate it in the air. A "control tower" regulates the traffic of airplanes around an airport, etc. Thus, the semantic characteristics of the constituent "control" are different in each of these sequences.

In conclusion, words such as "control" or "group" potentially correspond to dozens of concepts. Somehow, combining these two ambiguous words produces an unpredictable semantic "accident", that selects one specific, unambiguous concept among the hundred potential ones. In order to correctly process these combinations as unambiguous, we must describe them explicitly. In effect, this description -- naturally implemented in a NooJ dictionary -- is equivalent to processing them as multi-word units.

Terminological usage: if, to refer to a certain concept or object, people always use one specific sequence of words, the sequence must be lexicalized; it is therefore treated as a multi-word unit.

Fluent speakers often share a single specific expression to refer to each of numerous objects or concepts in our world environment. For instance, the meaning of the compound adjective "politically correct" could be as well expressed by dozens of expressions, including "socially respectful", "derogatorily free", "un-offensive to any audience's feelings", or, why not even "ATAM" for "Acceptable To All Minorities". However, there is a shared consensus among English speakers on how to express this

concept, and any person who does not use the standard expression would "sound funny".

Numerous noun phrases are used in a similar way. Compare, for instance, the following noun phrases to the left and right:

a shopping mall	*a commercial square, a retail district, a shop place*
a washing machine	*a cloth washer, an automatic cleaner, a textile cleaning device*
a health club	*a health saloon, a musculation club, a gym center*
a word processor	*a word typing program, a software writer, a text composer*
a parking lot	*a car area, a car square, a vehicle park, a resting car lot*
an ice cream	*a creamed ice, a sugary ice, an icy dessert, a cool sweet*

There is no morphological, syntactic, or semantic reason why the phrases to the left are systematically used, while none of those to the right is ever used. From a linguistic point of view, the *noun phrases* to the right are potential constructs that follow simple English syntactic rules but never occur in fact, whereas the *terms* to the left are elements of the vocabulary shared by English speakers, and hence must be treated as Atomic Linguistic Units (i.e. lexicalized multi-word units).

Illusion of productivity

One problem that has often misled linguists is that terms often exist in series that are apparently productive. For instance, consider the following list of terms constructed with the noun "insurance":

Automobile insurance, blanket insurance, fire insurance, health insurance, liability insurance, life insurance, travel insurance, unemployment insurance

The fact that there are a potentially large series of insurance terms may lead some linguists to assume that these terms are productive and therefore should not be listed nor described explicitly. But the alternative to an explicit description of these terms would be to use general syntactic rules, such as "any noun can be followed by 'insurance'." These rules would be useless if one wanted to distinguish the previous terms from the following sequences:

Anti-damage insurance, cover insurance, combustion insurance, sickness insurance, responsibility insurance, death insurance, tourism insurance, job loss insurance

In fact, each term from the former series can be linked to some description that would define its properties or meaning (e.g. a legal contract), whereas none of the sequences from the latter series actually has any precise meaning.

Properties definition file

In NooJ dictionaries (just like in INTEX's DELA-type dictionaries), users can create new information codes, such as **+Medic**, **+Politics** or **+transitive** and add them to any lexical entry. These codes become instantly available and can be used right away in any NooJ query or grammar, by using lexical symbols such as `<N+Medic>` or `<N+Hum-Politics+p>`, etc.

NooJ users can also use information codes that are in the form of a property with a value, such as in "**+Nb=plural**" or "**+Domain=Medic**". This allows NooJ to link properties and their values, as can be shown in the table view of the dictionary (**DICTIONARY > View as table**):

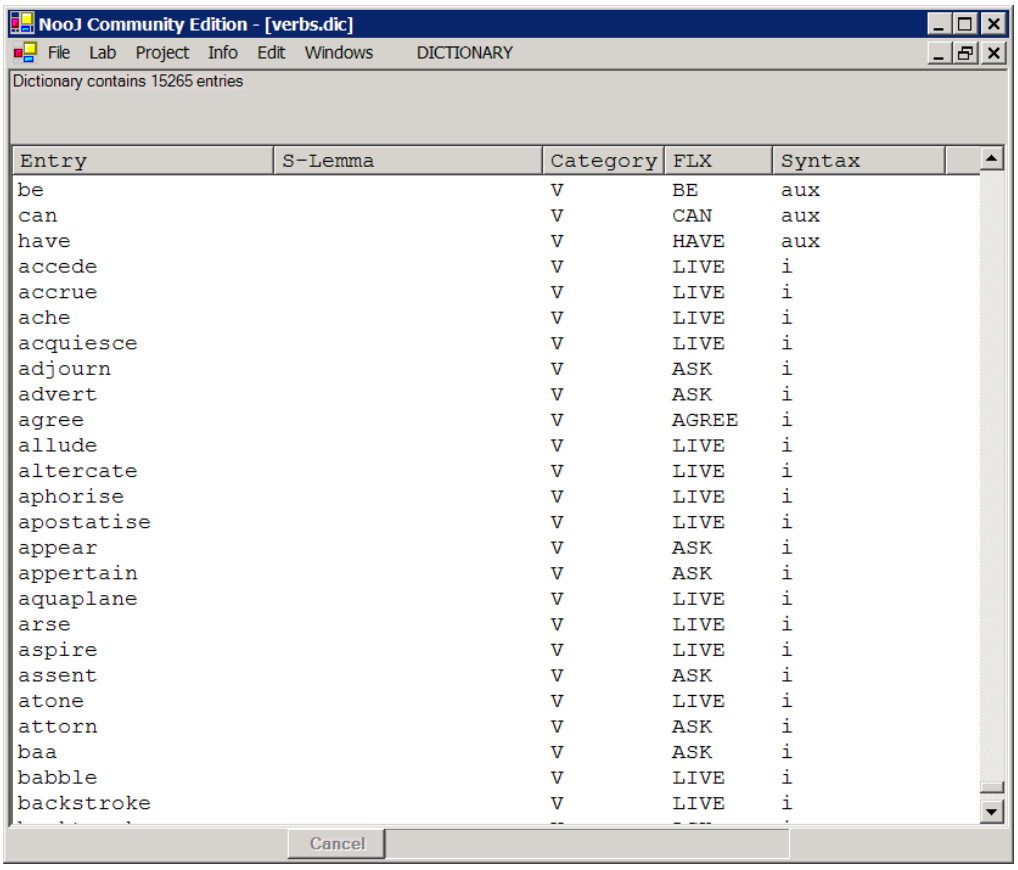


Figure 29. Displaying a dictionary as a table

In this dictionary, the **FLX** and the **Syntax** properties are associated with values, such as "+FLX=LIVE" or "+Syntax=aux".

It is also possible to define the relationship between categories, properties and their values by using a Dictionary Properties' Definition file. In this file, users can describe what properties are relevant for each morpho-syntactic category, and what values are expected for each property. The following figure shows a few properties' definitions:

```
V_Pers = 1 | 2 | 3 ;
V_Nb = s | p ;
```

```
V_Tense = G | INF | PP | PR | PRT ;
V_Syntax = aux | i | t ;
```

These definitions state that the category "**V**" is associated with properties "**Pers**", "**Nb**", "**Tense**" and "**Syntax**"; these properties's possible values are then listed. If these definitions are used, then the notation "**+Syntax=aux**" can be abbreviated into "**+aux**".

Finally, it is possible to let NooJ know that certain property values are *inflectional features*. That allows NooJ's morphological parser to be able to analyze a wordform by making it inherit its inflectional (or not inflectional) properties, from another wordform. For instance, we can tell NooJ's morphological parser to analyze the wordform "*disrespects*" by copying all the inflectional features of the wordform "*respects*" (i.e. +PR+3+s). To let NooJ knows what the inflectional features, we enter a single rule in the dictionary properties' definition file:

```
INFLECTION = 1 | 2 | 3 | m | f | n
           | s | p | G | INF | PP | PR | PRT ;
```

10. Inflection and Derivation

In order to analyze texts, NooJ needs dictionaries which house and describe all of the words of that text, as well as some mechanism to link these lexical entries to all the corresponding (inflected and/or derived) forms that actually occur in texts.

Inflected forms: in most languages, words are inflected as they are conjugated, used in the plural, in the accusative, etc. For instance, in English, verbs are conjugated, nouns are inflected in the plural. In French: verbs are conjugated, nouns and adjectives are inflected in the feminine and in the plural.

Derived forms: in most languages, it is possible to use a word, with a combination of affixes (prefixes or suffixes) to construct another word that is not necessarily of the same syntactic category. For instance, from the verb "to mount", we can produce the verb "to dismount", as well as the adjectives "mountable" and "dismountable"; from the adjective "American", we produce the verb "to americanize", etc.

Lexical entries in NooJ dictionaries can be associated with a paradigm that formalizes their inflection, i.e. verbs can be associated with a conjugation class, nouns can be associated with a class that formalizes how to write them in the plural, etc.

The aim of describing the inflection of a lexical entry is to be able to automatically link all its wordforms together, so that, for instance, the lexical symbol:

```
<be>
```

matches any of the forms in the set:

```
be, am, is, are, was, were, been, being
```

Note that lexical symbols do not define equivalence sets, because of potential ambiguities. For instance, the lexical symbol `<being>` matches all the previous

wordforms, as well as the wordform "beings" (because *being* is ambiguous, it belongs to two different sets of wordforms).

Inflectional Morphology

NooJ's inflection module is triggered by adding the special property "**+FLX**" to a lexical entry. For instance, in the _Sample.dic dictionary (file stored in "Nooj\en\Lexical Analysis"), we see the following entries:

```
artist,N+FLX=TABLE+Hum
cousin,N+FLX=TABLE+Hum
pen,N+FLX=TABLE+Conc
table,N+FLX=TABLE+Conc
man,N+FLX=MAN+Hum
```

This sample of a dictionary states that lexical entries *artist*, *cousin*, *pen* and *table* share the same inflectional paradigm, named "**TABLE**", while the lexical entry man is associated with the paradigm "**MAN**".

NooJ provides two equivalent tools to describe these inflectional paradigms: either graphically or by means of (textual) rules.

Both descriptions are equivalent, and internally compiled into a Finite-State Transducer (in ".nof" files).

Describing inflection graphically

The dictionary must contain at least one line that starts with a command such as:

```
#use NameOfAnInflectionalGrammar.nof
```

Paradigm names correspond to graphs included in the inflectional/derivational grammar. For example, for the following lexical entry:

```
cousin,N+FLX=TABLE+Hum
```

there must exist, in the inflectional grammar of the dictionary, a graph called "**TABLE**" which describes the two forms *cousin* and *cousins*. All of the nouns inflected in this manner (e.g. *artist, pen, table*, etc.) are associated with the same graph.

Inflectional paradigms can be described by transducers that contain as inputs, the suffixes that must be added to the lexical entry (i.e. lemma) in order to obtain each inflected form, and as outputs, the corresponding inflectional codes ("s" for singular, and "p" for plural). For example, here is the graph **TABLE** associated to nouns such as *cousin*:

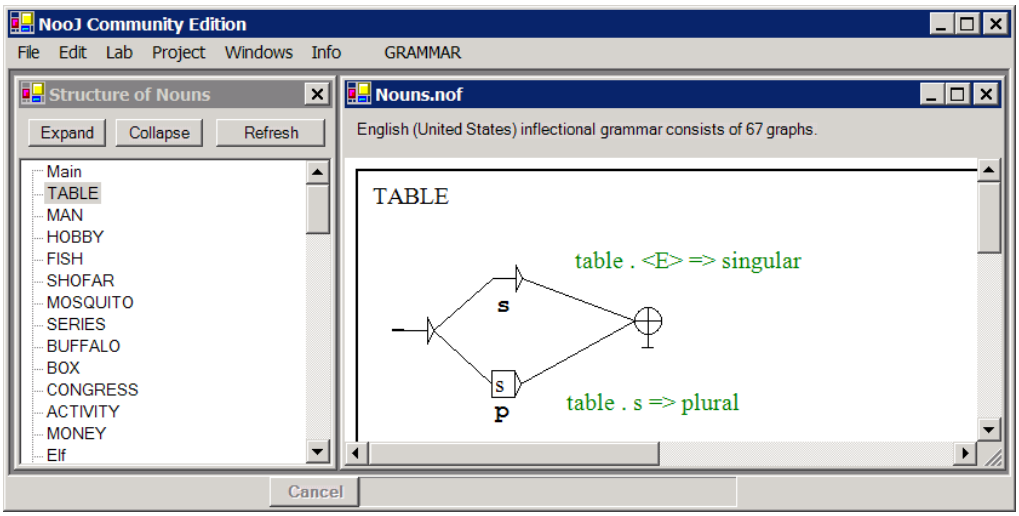


Figure 30. The graph for paradigm *TABLE*

This graph is used in the following manner:

-- (upper path) if we add the empty suffix to the lexical entry, we produce the form "cousin" associated with the inflectional codes "s" (singular);

-- (lower path) if we add the suffix "s" to the lexical entry, we produce the form "cousins" associated with the inflectional code "p" (plural).

Special operators

In English, numerous lexical entries are not merely prefixes of all their inflected forms; for example, the entry "*man*" is not a prefix of the form "*men*". In order to obtain that form from the lemma, we must delete the two last letters "*n*" and "a" of the lemma, and then add the suffix "*en*".

In NooJ, to delete the last character, we use the operator **<B>** (*Backspace*) and its variant <B2> (to perform the operation twice):

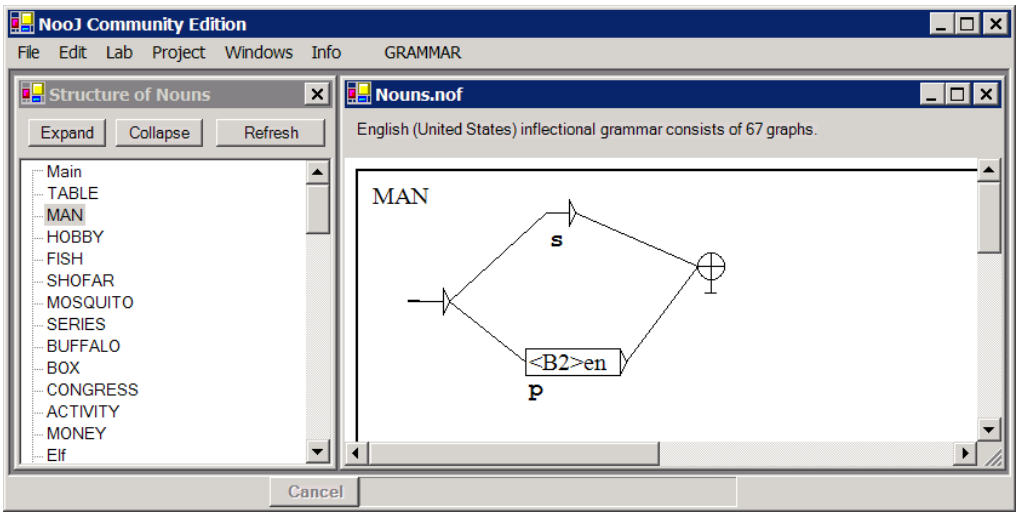


Figure 31. Inflectional paradigm *MAN*

Inflecting the lemma into its plural form is performed in three steps:

```
man.<B2>en → m.en → me.n → men
```

where the dot represents the top of a stack (i.e. the current cursor position). Each operation (either delete a character or add a new one) takes a constant time to run. Therefore, each inflected form can be computed in a time proportional to its length (Computer scientists use the term O(n) to describe the efficiency of an algorithm that runs in a time proportional to the length *n* of its input).

Thanks to the **<B>** operator, we can represent all possible types of inflection, including those considered more "exotic"; for example:

```
recordman.<B3>woman → recordwoman
```

However, NooJ understands 10 other operators:

<E>	empty string
<B>	delete last character
<D>	duplicate last character
<L>	go left
<R>	go right
<N>	go to the end of next wordform
<P>	go to the end of previous wordform
<S>	delete next character

For instance, in order to describe the plural form of the compound noun *bag of tricks*, we could perform the following operation:

```
bag of tricks.<P2>s → bags of tricks
```

(go to the end of the previous wordform twice, then add an "s").

Users can modify the behaviour of these operators, and add their own. For instance, the operators **<A>** (remove accent to the current letter), **<Á>** (add an acute accent) and **<À>** (add a grave accent) were added to most Romance languages, the operator **<F>** ("finalize" the current letter) was added to Hebrew, and the behaviour of the **<B>** command in Hebrew takes into account silent "e" when performed on a final consonant.

Use of Embedded graphs

In an inflectional grammar, it is possible to use graphs that will be embedded in one or more inflectional paradigms. This feature allows linguists to share a number of graphs that correspond to identical sets of suffixes, and also to generalize certain morphological properties. For instance, the following French graph represents the inflectional paradigm of the noun "cousin":

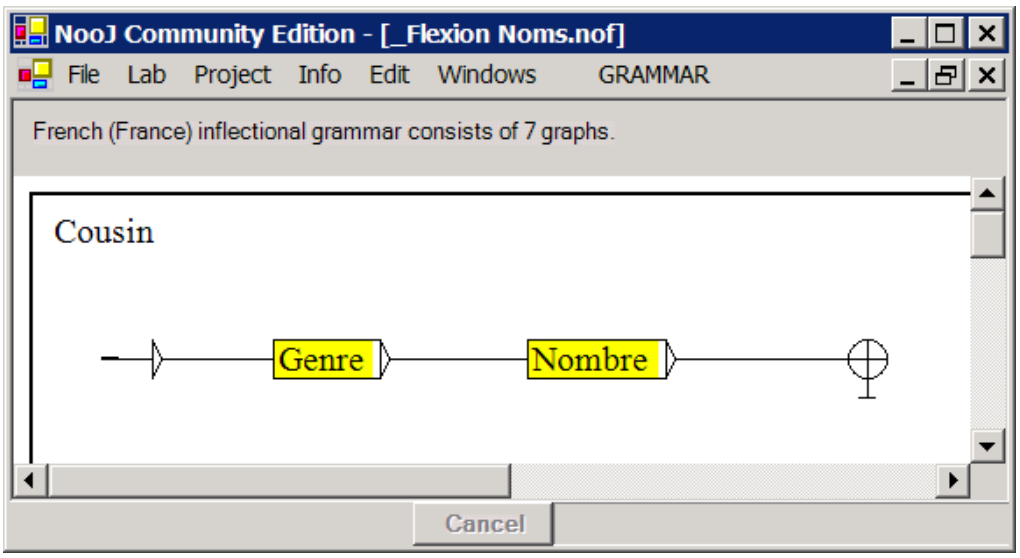


Figure 32. Inflectional paradigm with embedded graphs

the embedded graph "Genre" takes care of the gender suffix (add an "e" for feminine) and the graph "Nombre" takes care of the number suffix (add an "s" for plural):

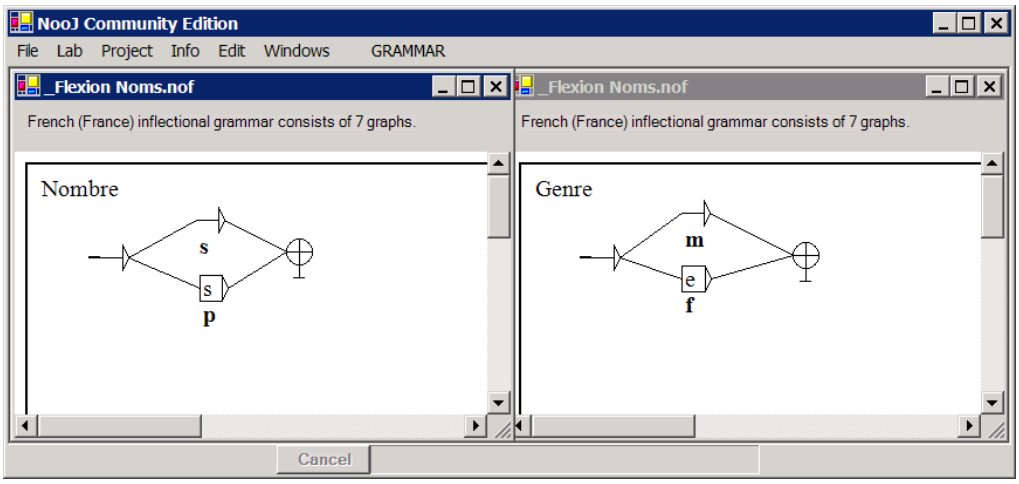


Figure 33. The two embedded graphs

In the same manner, one could design a French graph "Future" that would represent the set of suffixes: *ai*, *as*, *a*, *ons*, *ez*, *ont*, and simply add this graph to a number of verb conjugation paradigms.

Describing Inflection textually

The dictionary must contain at least one line that starts with a command such as:

```
#use NameOfAnInflectionalDescriptionFile.nof
```

Paradigm names correspond to the rules included in the inflectional description file. For example, for the following lexical entry:

```
help,V+FLX=ASK
```

there must exist, in the inflectional description file of the dictionary, a rule called "**ASK**" which describes the all conjugated forms of the verb *to help*. All of the verbs that conjugate in this manner (e.g. *love, ask, count*, etc.) are associated with the same rule. Below is the rule "**ASK**":

```
ASK = <E>/INF | <E>/PR+1+2+s | <E>/PR+1+2+3+p |
s/PR+3+s | ed/PP | ed/PRT | ing/G ;
```

This paradigm states that if we add an empty string to the lexical entry (e.g. *help*), we get the infinitive form of the verb (*to help*), the Present, first person or the second person singular (*I help*), or any of the plural forms (*we help*). If we add an "s" to the entry, we get the Present, third person singular (*he helps*). If we add "ed", we get the past participle form (*helped*) or any of the preterit forms (we helped). If we add "ing", we get the gerundive form (*helping*).

Never forget to end each rule definition with a semi-colon character ";".

Use of Embedded rules

Just as it is possible to embed graphs in an inflectional/derivational grammar, it is also possible to add auxiliary rules that do not correspond directly to paradigm names, but can be used by other rules. For instance, in the following French inflectional description file:

```
Genre = <E>/m | e/f;
Nombre = <E>/s | s/p;

Crayon = <E>/m :Nombre;
Table = <E>/f :Nombre;
Cousin = :Genre :Nombre;
```

The two rules **Genre** and **Nombre** are auxiliary rules that are shared by the three paradigms **Crayon**, **Table** and **Cousin**.

Inflecting Multiword units

Multiword units which inflection operates on the last component are inflected exactly in the same manner as simple words. For instance, to inflect the term "jet plane", one can use the same exact rule as the one used to inflect the simple noun "plane":

```
PEN = <E>/singular | s/plural;
```

However, when other components of a multiword unit inflect, such as in "man of honor", NooJ provides the two operators **<N>** (go to the end of the next wordform) and **<P>** (go to the end of the previous wordform) in order to inflect a selected component:

```
MANOFHONOR = <E>/singular | <PW><B2>en/plural;
```

The operator **<PW>** moves the cursor to the end of the first component of the multiword expression, i.e. "man". The operator **<B2>** deletes the two last letters of the wordform, then the suffix **en** is added to the wordform; the resulting plural form is "men of honor".

Note that one could have used the **<P2>** operator (go to the previous wordform, twice) to do exactly the same thing. However, the operator <PW> is more general, and our paradigm works the same, regardless of the length of the multiword unit, e.g. "man of the year", "man of constant sorrow", etc.

Another, less direct but more powerful method is to reuse the paradigms of simple words in order to describe the paradigms of multiword units. For instance, the inflection of "man of honor" could be described by the two rules:

```
MAN = <E>/singular | <B2>en/plural;
MANOFHONOR = <PW> :MAN;
```

The first paradigm is used to inflect the simple noun "man"; the second paradigm reuses the first one to inflect all the multiword units that start with "man".

When inflecting multiword units in which more than one component needs to be inflected at the same time, there are two cases:

(1) there is no agreement between the inflected components

For instance, in the multiword unit "man of action", one can find the four variants: "man of action", "man of actions", "men of action", "men of actions". In that case, the paradigm used to describe the inflection of the multiword unit is very simple: just insert the paradigms used to inflect each of the inflected components:

```
MANOFACTION = :PEN <PW> :MAN;
```

(1) the inflected components agree in gender, or in number, or in case, etc.

For instance, in the French multiword unit "cousin germain", the noun "cousin" and the adjective "germain" agree both in gender and in number. In that case, we describe the inflection of the multiword unit exactly as if the components do not agree, e.g.:

```
COUSIN = <E>/mas+sin | s/mas+plur | e/fem+sin |
es/fem+plur ;

COUSINGERMAIN = :COUSIN <PW> :COUSIN;
```

However, we tell NooJ to check the agreement between the components by checking the option "**Check Agreement**" in the Dictionary Compilation window:

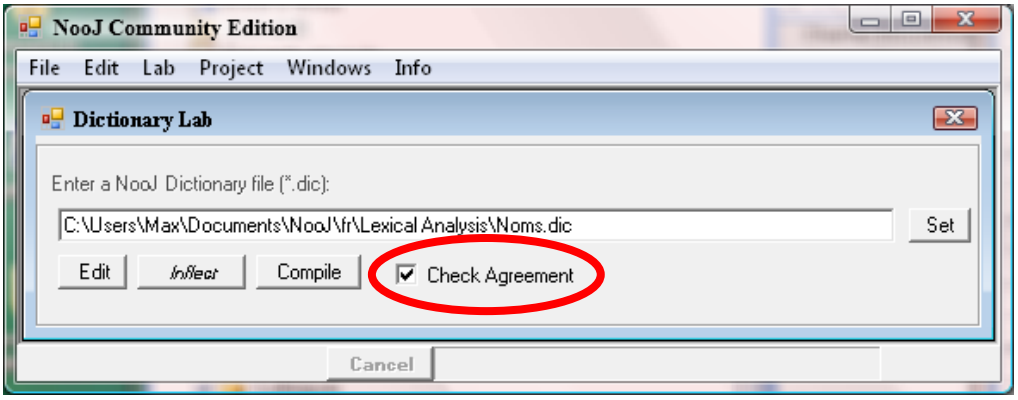


Figure 34. Agreement in multiword units

Finally, another possibility is to simply use the **COUSIN** paradigm directly in the **FLX** property in dictionary, e.g.:

```
cousin germain,N+FLX=COUSIN<P>COUSIN
```

Derivational Morphology

Inflectional grammars (both graphical and textual) can include derivational paradigms. Derivational paradigms are very similar to inflectional paradigms, except that the morpho-syntactic category of each wordform must be explicitly produced by derivational transducers. The special information "**+DRV**" is used to indicate a derivational paradigm.

For instance, here is a derivational description:

```
ER = er/N;
```

This rule states that by adding the suffix "er" to the lexical entry, one gets a noun. Now consider the following lexical entry:

```
laugh,V+DRV=ER:TABLE
```

This entry makes NooJ produce the noun "laugher", which is then inflected according to the paradigm "**TABLE**". The two forms "laugher" and "laughers" will then be lemmatize as "laugh", even though they will be associated with the category "N". In consequence, query symbols such as <laugh> will match both the conjugated forms of the verb "to laugh", as well as its derived forms, including the noun "a laugher", adjectives such as "laughable", etc.

Default inflectional paradigm

Often, forms that are derived from an initial lemma inflect themselves exactly like the initial lemma. For instance, the verb "to mount" can be prefixed as "dismount"; the latter verb inflects just like "to mount". In the same manner, the prefix "re" can be used to produce the verb "remount", that inflects just like "mount".

In these cases, it is not necessary to specify the inflectional paradigm to be applied to the derived form: just omit it. For instance:

```
mount,V+FLX=ASK+DRV=DIS
```

states that the verb "to mount" inflects according to the inflectional paradigm "**ASK**", and then derives according to the derivational paradigm "**DIS**". The latter paradigm produces the verb "dismount", which then inflects according to the default inflectional paradigm "**ASK**". If the derivational paradigm "**DIS**" is defined as:

```
DIS = <LW>dis/V;
```

the previous entry will allow NooJ to recognize any conjugated of the verbs "mount" and "dismount".

In the same manner, consider the following lexical entry

```
laugh,V+FLX=ASK+DRV=ABLE:A+DRV=ER:TABLE
```

This entry allows NooJ to recognize any conjugated form of the verb "to laugh", plus the adjective form "laughable", plus the two noun forms "laugher" and "laughers".

Inflectional / derivational graphical and textual grammars (.nof files) are equivalent: NooJ compiles both into the same Finite-State Transducers, which are interpreted by the same morphological engine. Graphs are typically used for small paradigms, such as the English conjugation system (where we have only a few wordforms per class), whereas textual rule-based descriptions are used for heavier inflectional systems, such as Verb conjugations in Romance Languages (30+ forms per verb).

Compile a dictionary

To make NooJ apply a dictionary during its linguistic analysis of texts and corpora, you need to compile it.

(1) Make sure that the inflectional / derivational grammars (".nof" files), as well as the dictionary's properties definition files (".def" files) are stored in the same folder as the dictionary (usually, in the Lexical Analysis folder of the current language).

(2) Make sure that you have added the commands to use the inflectional / derivational paradigms' definitions as well as the properties definitions before they are needed (preferably at the top of the dictionary), e.g.:

```
#use properties.def
#use nouns-inflection.nof
#use verbs-inflection.nof
#use nominalization.nof
```

(3) When the dictionary is ready, compile it by clicking **Lab > Dictionary > Compile**. This computes a minimal deterministic transducer that contains all the lexical entries of the original dictionary, plus all the corresponding inflectional and derivational paradigms, represented in such a way that all the wordforms associated with the lexical entries are both recognized and associated with the corresponding properties.

This transducer is **deterministic** because the prefixes of all lexical entries are factorized. For example, if several thousands of entries begin with an "a", the transducer contains only one transition labeled with letter "a";

This transducer is **minimal** because the suffixes of all wordforms are also factorized. For example, if tens of thousands of wordforms end with "*ization*", and are associated with information "*Abstract Noun*", this suffix, along with the corresponding information is only written once in the transducer.

Attention INTEX users: the compiled version of a NooJ dictionary is **NOT** equivalent to the transducer of a DELAF/DELACF dictionary. NooJ's compiled dictionaries contain all (inflectional and derivational) paradigms associated with their entries; this allows NooJ's parsers to be able to perform morphological operations, such as **Nominalization**, or **Passivation**. In essence, this new feature makes it possible to implement Transformational Analysis of texts.

(4) Check the compiled dictionary (a ".nod" file) in the Info > Preferences > Lexical Analysis, so that next time "Linguistic Analysis" is performed, the dictionary will be used. You might also want to give it a high, or low priority.

The menus **DICTIONARY** and **Lab > Dictionary** propose a few useful tools:

-- Sort the dictionary (**DICTIONARY > Sort**). NooJ's Sorting program respects the current language's and culture's alphabetical order. It processes comments and empty lines as zone delimiters. Users can then create zones in a dictionary, such as a zone for Nouns followed by a zone for Adjectives, etc. NooJ will then sort each zone independently.

-- Check the format of its lexical entries: **DICTIONARY > Check**. NooJ will make sure each lexical entry is in a valid format: one entry, one comma, one optional lemma, one category in uppercase latin non-accented letters, optional property features.

-- Check that all the properties and corresponding values that you have used in the dictionary are consistent with the Properties' definition file (you just need to display the dictionary as a table, and then check that all columns are correctly filled).

-- Check the inflection and derivation implemented by your dictionary, and display the resulting list of inflected and derived wordforms. To do so, Click **Lab > Dictionary > Inflect**.

11. Morphological Analysis

Wordforms are usually analyzed by a lookup of a dictionary associated with some optional inflectional or derivational description, as seen in the previous chapters. But there are cases where it is more natural to use productive morphological rules, rather than dictionaries, to represent sets of wordforms. Moreoever, morphological grammars can represent agglutinated sequences of ALUs properly. In NooJ, morphological rules are implemented inside morphological grammars, that are grammars which input recognizes the wordforms, and which output computes the corresponding linguistic information. Morphological grammars, just like dictionaries, are stored in the "Lexical Analysis" folder of the current language, and the results of the morphological parsing are stored, exactly as the results of the dictionaries' lookup, in the Text Annotation Structure.

A morphological grammar can be as simple as a graph that recognizes a fixed set of wordforms and associates them with some linguistic information. More complex grammars can recursively split wordforms into smaller parts -- their "affixes" -- and then enforce simple or more complex morpho-syntactic constraints to each affix.

Lexical Grammars

Lexical grammars -- i.e. with no constraints -- are simple transducers or RTNs that associate recognized wordforms with linguistic information. Generally, one uses these grammars to link together families of variants, or when the number of wordforms would be too large to be listed in a dictionary, whereas they can easily be represented by a few productive grammars.

A simple lexical grammar

The following lexical grammar can be found in "My documents\Nooj\en\Lexical Analysis". It is an example of an elementary grammar that recognizes a family of spelling variants:

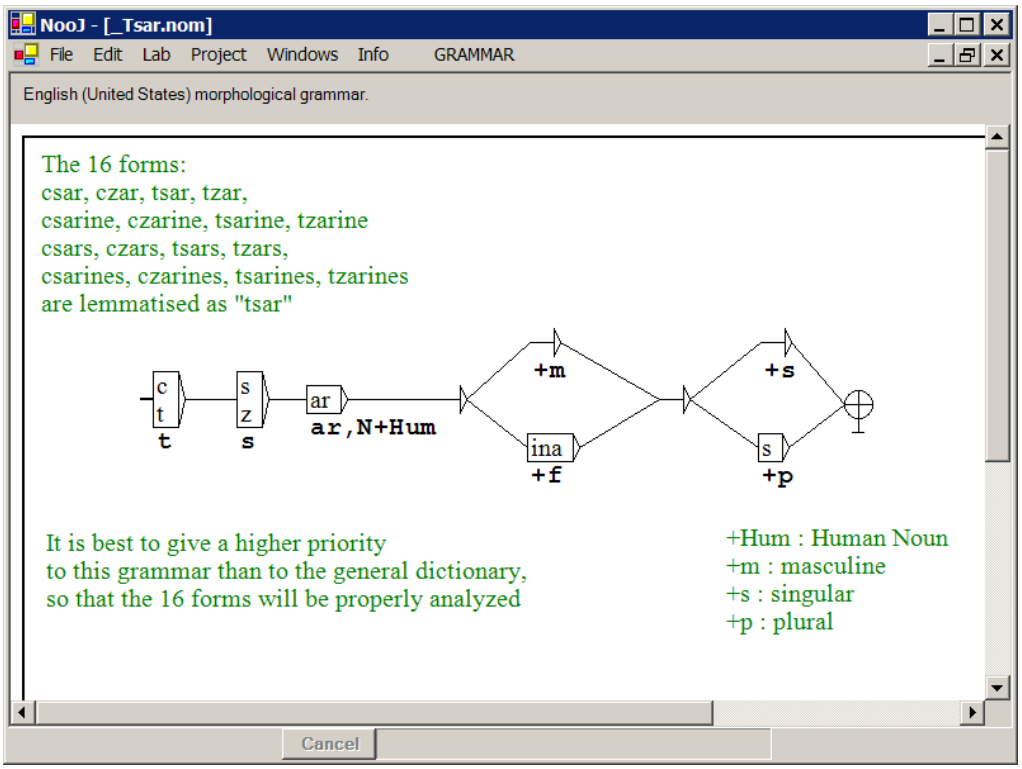


Figure 35. Morphological Grammar "tsar"

This graph recognizes sixteen variants of the wordform *tsar*, and associates them with the lemma *tsar* and the linguistic features "**N+Hum**" (*Noun, Human*), as well as their gender (**+m** or **+f**) and number (**+s** or **+p**).

To enter the output attached to a node, follow the label of the node with a slash character "/", and then the lexical information. There are comments displayed in green. To enter a comment, create a node that will not be connected to any other node. The arrows are nodes labeled with the empty string (<E>).

Some of the information is being computed "on the fly", i.e. along the path, during the recognition process. For instance, the suffix "ina" is locally associated with the linguistic feature **+f**.

This grammar is equivalent to the following dictionary:

```
csar,tsar,N+Hum+m+s
csarina,tsar,N+Hum+f+s
csarinas,tsar,N+Hum+f+p
csars,tsar,N+Hum+m+p
czar,tsar,N+Hum+m+s
czarina,tsar,N+Hum+f+s
czarinas,tsar,N+Hum+f+p
czars,tsar,N+Hum+m+p
```

```
tsar,tsar,N+Hum+m+s
tsarina,tsar,N+Hum+f+s
tsarinas,tsar,N+Hum+f+p
tsars,tsar,N+Hum+m+p
tzarina,tsar,N+Hum+f+s
tzarina,tsar,N+Hum+f+s
tzarinas,tsar,N+Hum+f+p
tzarinas,tsar,N+Hum+f+p
```

This grammar, and similar ones, can be used by software applications to associate "variants" of a term, whether orthographic, phonetic, synonymous, semantic, translation, etc., with one particular canonical form that acts as an index key, an hyperonym, or canonical representative, or a "super lemma". For instance, using the previous grammar, indexing a corpus would produce one single index key for the sixteen wordforms, which is much better than regular indexers that typically builds several unrelated index keys for *csar*, *czar*, *tsar* and *tzar*. Moreover, NooJ's lexical symbol `<tsar>` now would match these sixteen wordforms in texts.

Sets of synonymous terms (such as WordNet's Synsets) can be represented via this feature. NooJ then becomes a search engine capable of retrieving synonymous words, e.g. the query `<germ>` matches all its "family's members" such as *bacteries, decease, sickness,* etc. as well as all its translations, if WordNet dictionaries for different languages are aligned.

Roman numerals

Here is a simple example of a morphological grammar that recognizes roman numerals:

```
I, II, III, IV, …, CXXXIX …
```

It is out of the question to create a dictionary containing all of the roman numerals (here we arbitrarily stop at 3,999). Rather, we create a morphological grammar that contains four graphs to represent the units, the tens, the hundreds and the thousands, as well as a main graph to bring all the graphs together.

The following morphological grammar can be found in "My documents\Nooj\en\Lexical Analysis".

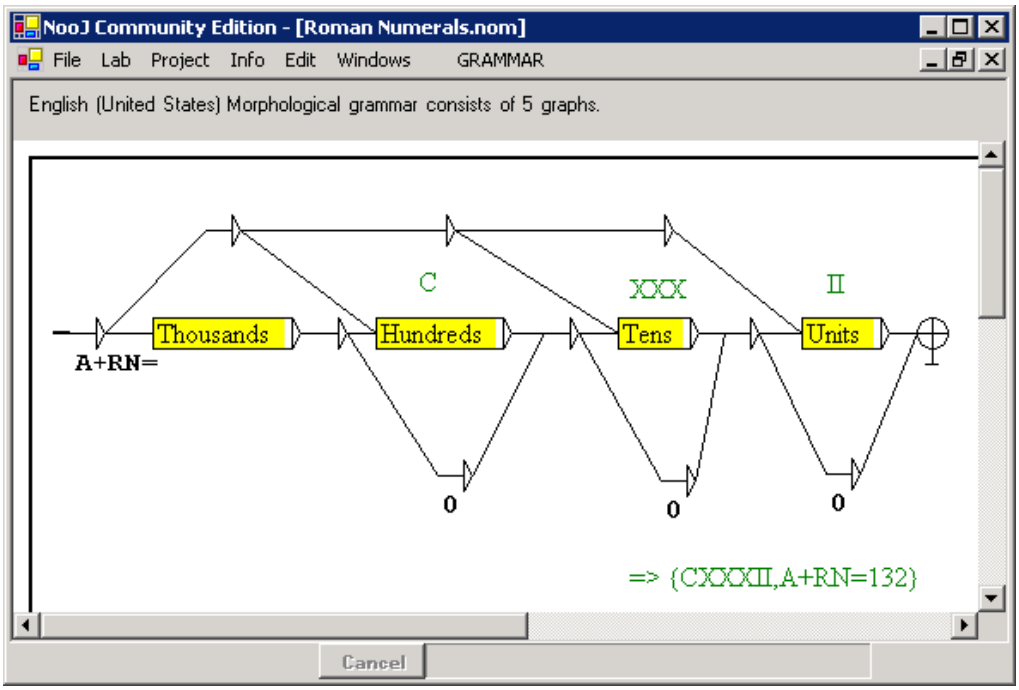


Figure 36. Roman numerals, main graph

This grammar recognizes and analyses Roman Numerals from **I** (1) to **MMMCMXCIX** (3,999), such as "CXXXII", and then tag them. The resulting tags look like:

```
CXXXII,A+RN=132
```

i.e. exactly as if this line had been explicitly entered in a dictionary. In order to do that, the grammar produces the output "**A+RN=**" followed by the numeric value of the roman numeral.

For example, in the main graph displayed above, the initial node is labeled as:

<E>/A+RN=

and the three <E> nodes (displayed as arrows) at the bottom of the graph, that are used to skip the "Hundreds", the "Tens" and the "Units", are naturally labeled with:

<E>/0

This morphological grammar is more complex than the previous "tsar" one, because it contains references to embedded graphs (displayed as *auxiliary nodes* in yellow).

NooJ grammars are organized sets of graphs. Each graph can include auxiliary nodes, that are references to other graphs. This recursive mechanism makes NooJ grammars equivalent to Recursive Transition Networks.

To enter an auxiliary node (displayed in yellow), prefix the label of the node with a colon character ":", and then enter the name of the embedded graph.

For example, in the main graph displayed above, the node Units is labeled as ":Units". Display it (**Alt-Click** an auxiliary node to explore its content):

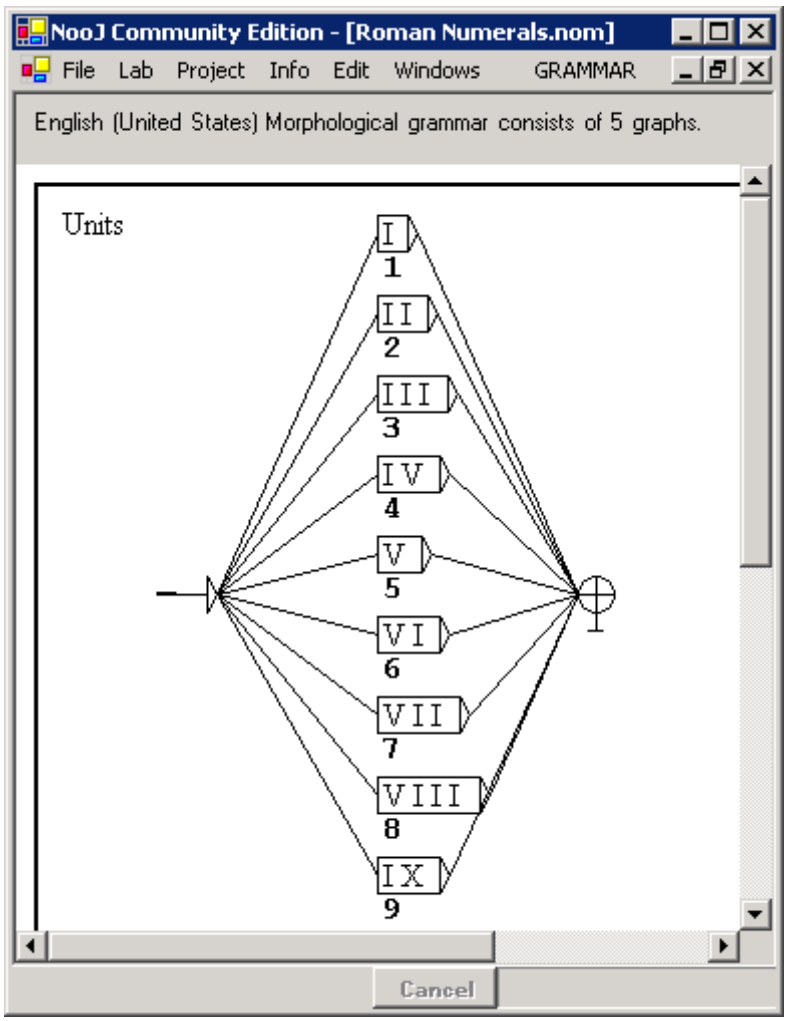


Figure 37. Roman numerals, the units

Notice that each path implements a local translation of a roman numeral (e.g. "VII") with its corresponding Arabic number ("7").

You can navigate in a grammar's structure, either by pressing the following keys:
-- "**U**" to display the current graph's parent,
-- "**D**" to display the current graph's first child,
-- "**N**" to display the next child,
-- "**P**" to display the previous child,
-- "**Alt-Click**" an auxiliary node to display it.
or by displaying the grammar's structure in a window (**GRAMMAR > Show Structure**).

The Tens graph is shown below:

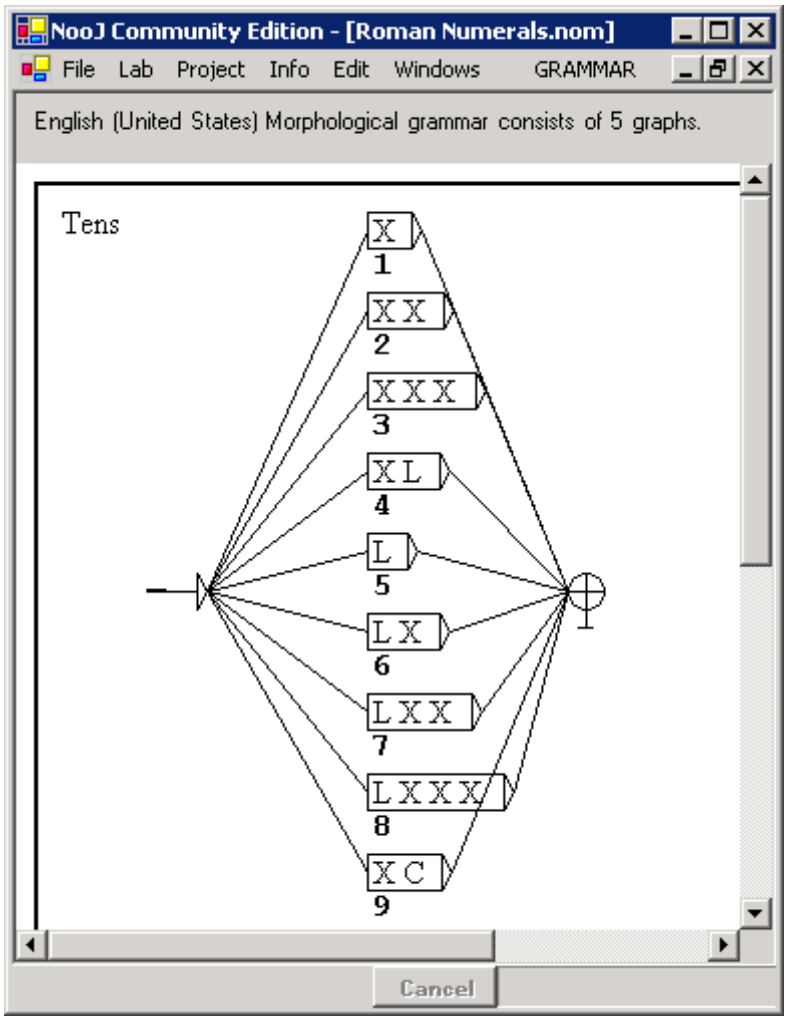


Figure 38. Roman numerals, the tens

In the same way, the graph that represents the "hundreds" is similar to the previous ones: just replace the "X" symbols with C's and L's with D's. Finally, the graph that represents the "Thousands" is displayed below:

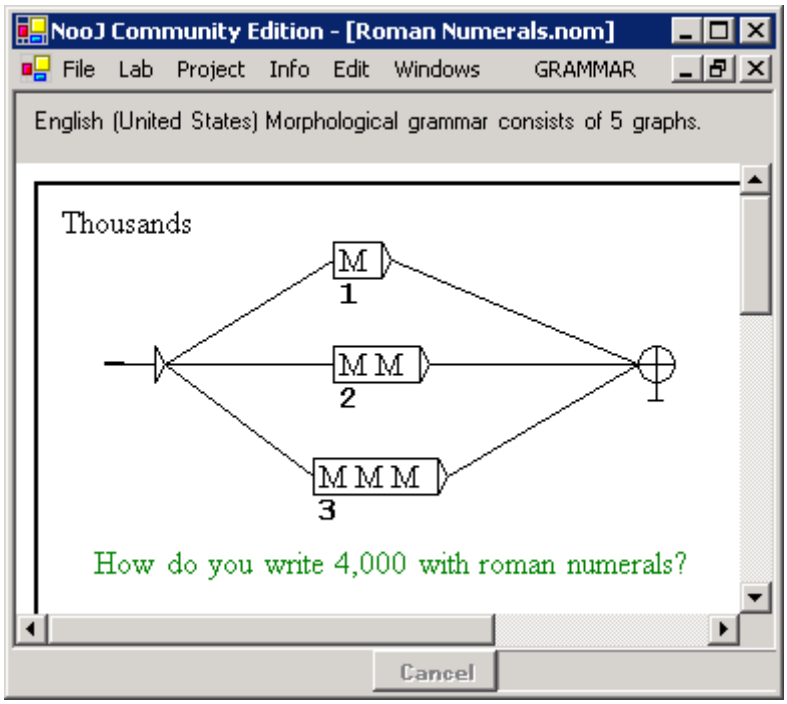


Figure 39. Roman numerals: the thousands

You can check this grammar by entering a few valid roman numerals in the grammar's contract (**GRAMMAR > Show Contract**):

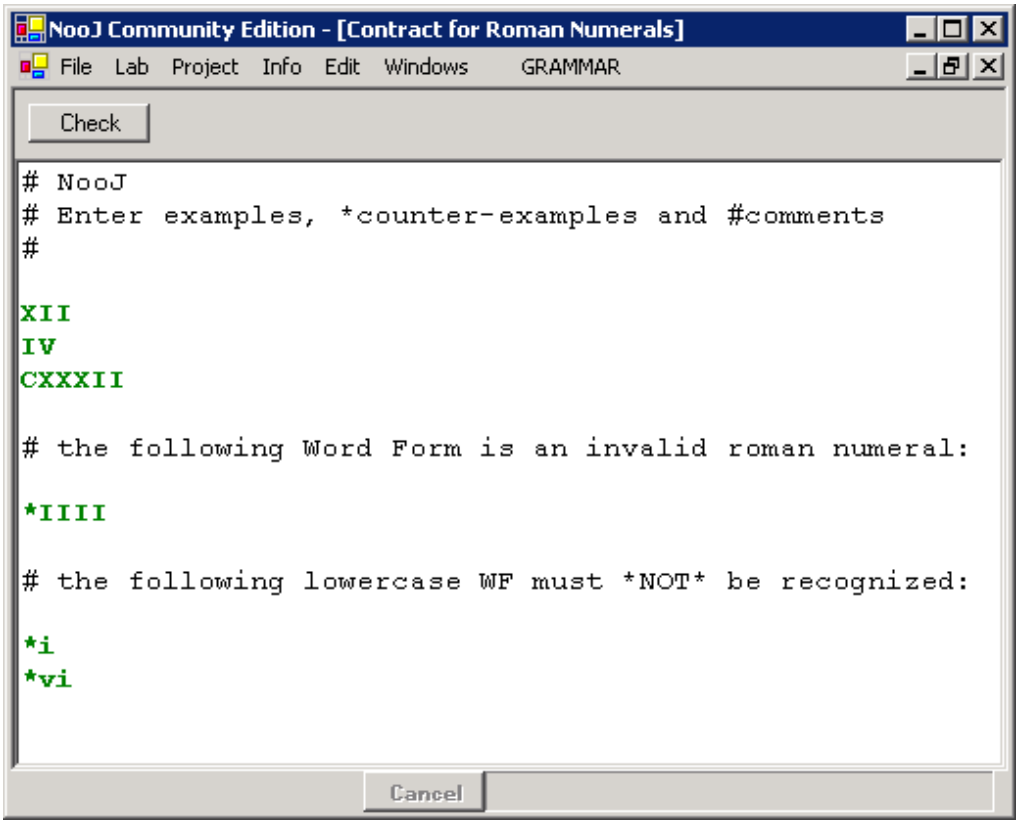


Figure 40. A grammar's contract

Make sure that valid roman numerals are indeed recognized by clicking the Check button. On the other hand, you can enter counter-examples, i.e. wordforms that are **not** roman numerals, e.g. "iiii"; in that case, prefix them with a star character "*" to tell NooJ that they must *NOT* be recognized.

The grammar's contract is a series of examples (i.e. words or expressions that MUST be recognized by the grammar, as well as counter-examples, i.e. words or expressions that MUST NOT be recognized by the grammar. When saving a grammar, NooJ always check that its contract is honored.

When developing complex grammars, i.e. grammars with embedded graphs, that you intend to use for a long time, or that you will share with others, always use contracts: contracts guarantee that you will never break a grammar without noticing!

A simple grammar for unknown words

The two previous grammars (**tsar** and **Roman Numerals**) are equivalent to dictionaries, i.e. they recognize and tag a finite set of wordforms that could also be listed extensively in a dictionary. NooJ morphological grammars can also represent infinite sets of wordforms.

For instance, here is a (rather naïve) grammar that implements a Proper name recognizer:

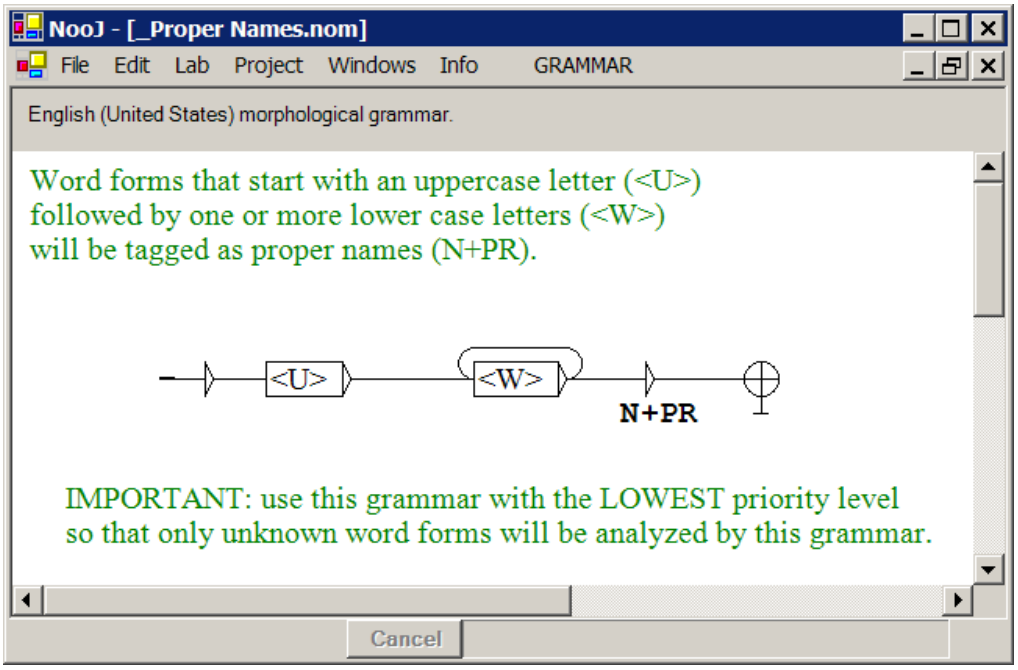


Figure 41. Naïve Proper Name recognition

The symbol **<U>** matches any uppercase letter; the symbol **<W>** matches any lowercase letter. Hence, this graph recognizes all the wordforms that begin with an uppercase letter, followed by one or more lowercase letters. For instance, the grammar matches the wordforms "Ab" and "John", but not "abc", "A" or "INTRODUCTION". All recognized wordforms will be associated with the corresponding linguistic information, i.e. "**N+PR**" (*Noun*, *Proper Name*).

NooJ's morphological grammars use 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

The previous grammar recognizes an infinite set of wordforms, such as *John* and *The* (for instance, when this wordform occurs at the beginning of a sentence). Note that when a wordform is recognized, it is processed as if it was an actual entry of a dictionary, for instance:

```
John,N+PR
The,N+PR
```

It is best to give morphological grammars that implement productive rules a low priority (see next chapter), to make sure that only wordforms that were **not** recognized by other dictionaries or grammars are analyzed.

Computing lemma and linguistic information

In the previous examples, the lemma associated with the wordforms to be analyzed was either explicitly given (e.g. "tsar"), or implicitly identical to the wordforms to be recognized. In the same manner, the linguistic information to be associated with the recognized wordforms, i.e. "N+Hum" or "A+RN=132" was explicit in the output of a graph.

It is also possible to compute the resulting lemma and/or bits of the resulting linguistic information. To do that, one must explicitly produce the actual linguistic unit as a NooJ annotation (in NooJ, annotations are internally represented between angles "<" and ">"). This explicit notation triggers several functionalities, including the capability of assembling the final annotation along the path, the capability to produce more than one annotation for a word sequence, and the use of variables to store affixes of the recognized sequence.

Consider the following grammar:

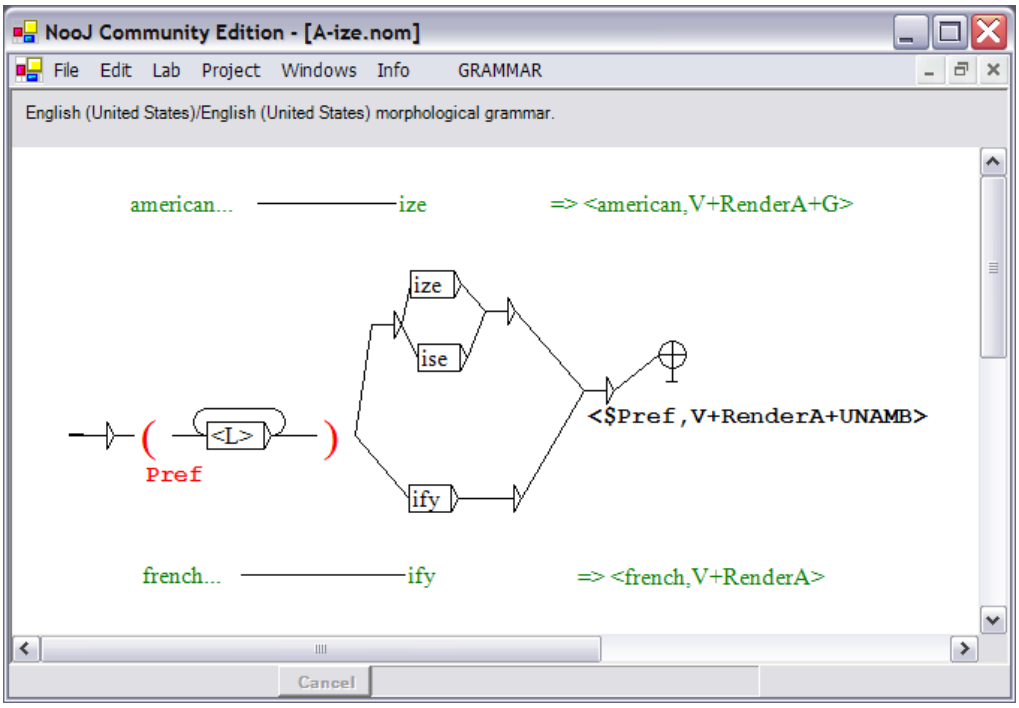


Figure 42. Removing the suffix "-ize"

This grammar recognizes any wordform that ends with "ize", "ise" or "ify" (the loop with **<L>**'s matches any sequence of letters). During the recognition process, the prefix (i.e. the beginning letters) is stored in the variable **$Pref**.

To store an affix of a sequence in a variable while parsing the sequence, insert the special nodes "**$(**" and "**$)**" around the affix. You need to name the variable: in order to do so, add the variable's name behind the opening parenthesis, e.g. "**$(VariableName**". Variables' nodes appear in red by default.

Recognized wordforms are then associated with the corresponding annotation:

```
<$Pref,V+RenderA>
```

in which the variable **$Pref** is replaced with its value. For instance, when the wordform "americanize" is recognized, the variable **$Pref** stores the value "american"; the wordform is then associated with the following annotation:

```
<american,V+RenderA>
```

i.e. as if one of NooJ's dictionaries contained the following lexical entry:

```
americanize,american,V+RenderA
```

Similarly, the wordform "frenchify" will be tagged as `<french,V+RenderA>`.

Morphological grammars produce a superlemma that can be used in any of NooJ queries of syntactic grammars; for instance, the symbol <american> will now match *American* and *Americans* as well as *americanize*.

Note that unfortunately, the previous "naïve" grammar also matches wordforms such as "size", and then produces an incorrect analysis:

<s,V+RenderA>

We will see how to fix this problem later.

By splitting wordforms into several affixes stored in variables, it is possible to process complex morphological phenomena, such as the deletion or addition of letters at the boundaries between affixes. For instance, the following graph could be used to process the French 'ism' suffix, when used after the name of a political figure:

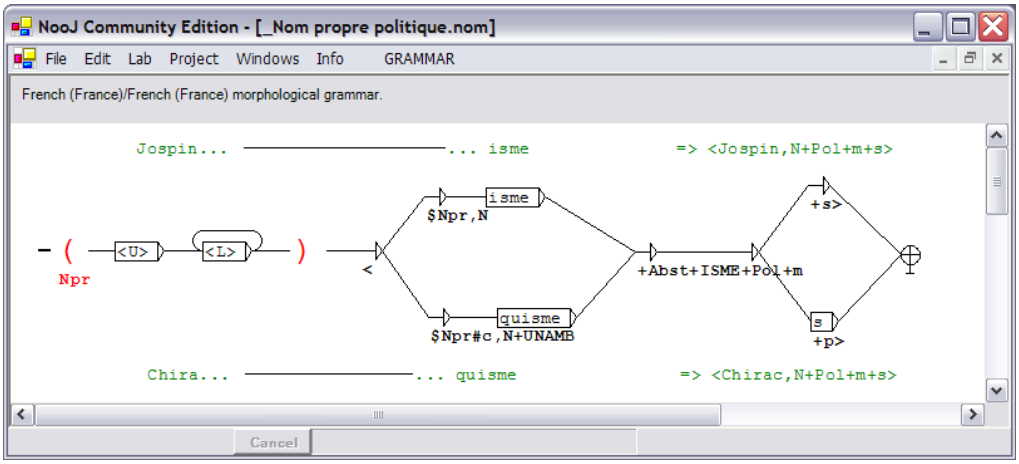


Figure 43. Managing affix boundaries

When the wordform "Jospinisme" is recognized, variable **$Npr** stores the prefix "Jospin". The resulting annotation becomes:

<Jospin,N+Abst+ISME+Pol+m+s>

When the wordform "Chiraquisme" is recognized, variable **$Npr** holds the prefix "Chira". The resulting annotation is:

<Chirac,N+UNAMB+Abst+ISME+Pol+m+s>

Notice that the lemma here is produced by concatenating the value of **$Npr** and a final "c": **$Npr#c**. As with dictionary entries, the special code **+UNAMB** disables any other analyses, so that for the wordform "Chiraquisme" is not recognized by the top path as well (that would produce the incorrect lemma "Chiraqu").

One wordform represents a sequence of more than one annotation

Morphological grammars can also produce more than one annotation for a particular wordform. This feature allows one to process contracted words, such as the wordform "*cannot*" or the French contracted word "du". The following grammar for instance associates "cannot" with a sequence of two annotations:

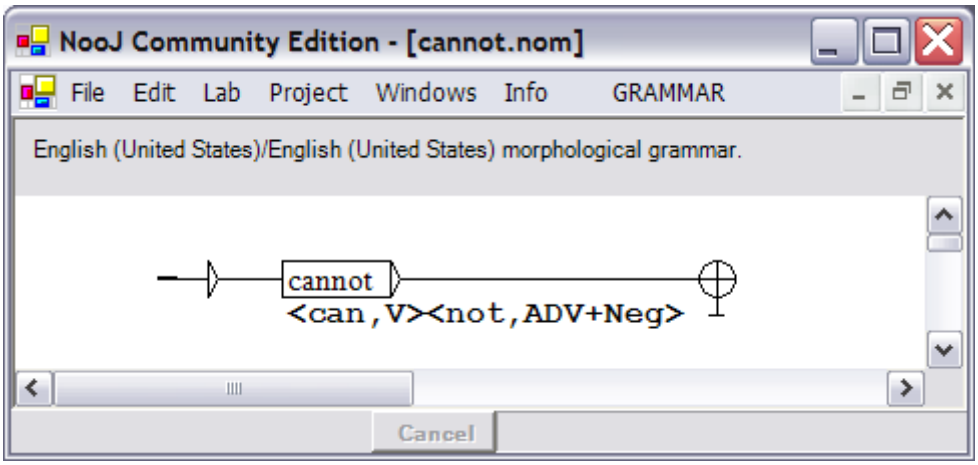


Figure 44. Contracted words

Note that if this resource is checked in Info > Preferences > Lexical Analysis, the linguistic analysis of texts will annotate the wordform cannot as a sequence of two annotations:

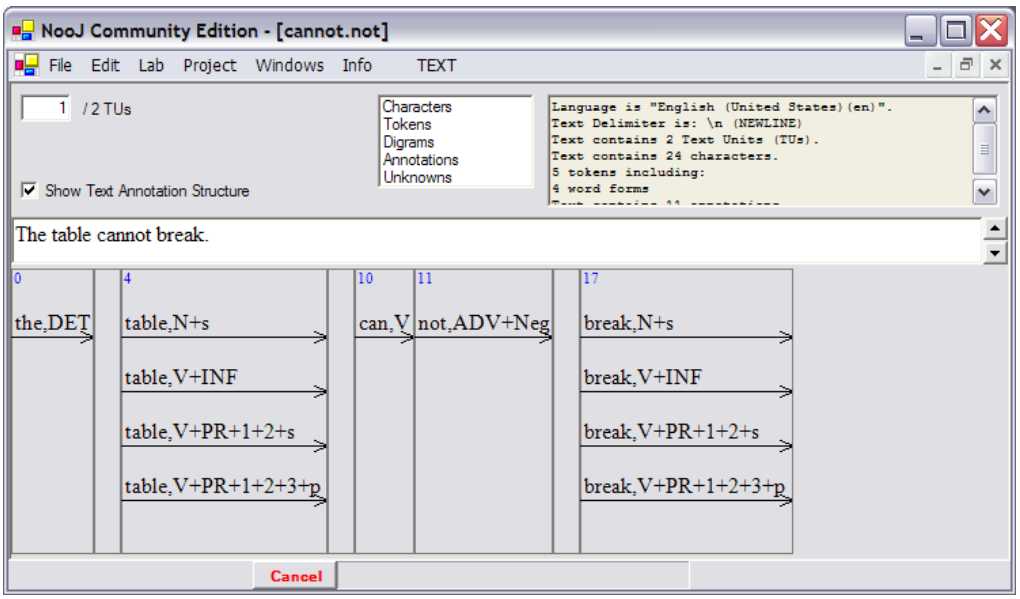


Figure 45. Annotation for contracted word "cannot"

The ability to produce more than one annotation for a single wordform is essential to the analysis of Asian and Germanic languages.

Repetitions

Variables can also be used in the input of a grammar to check for repetitions. For instance, the following grammar checks for letter repetitions in wordforms. Note in the contract that the wordforms "efficient" and "grammar" are indeed recognized, whereas the wordform "table" is not (the star is used to enter conter-examples in grammars' contract). All recognized wordforms will be annotated with the category "REP".

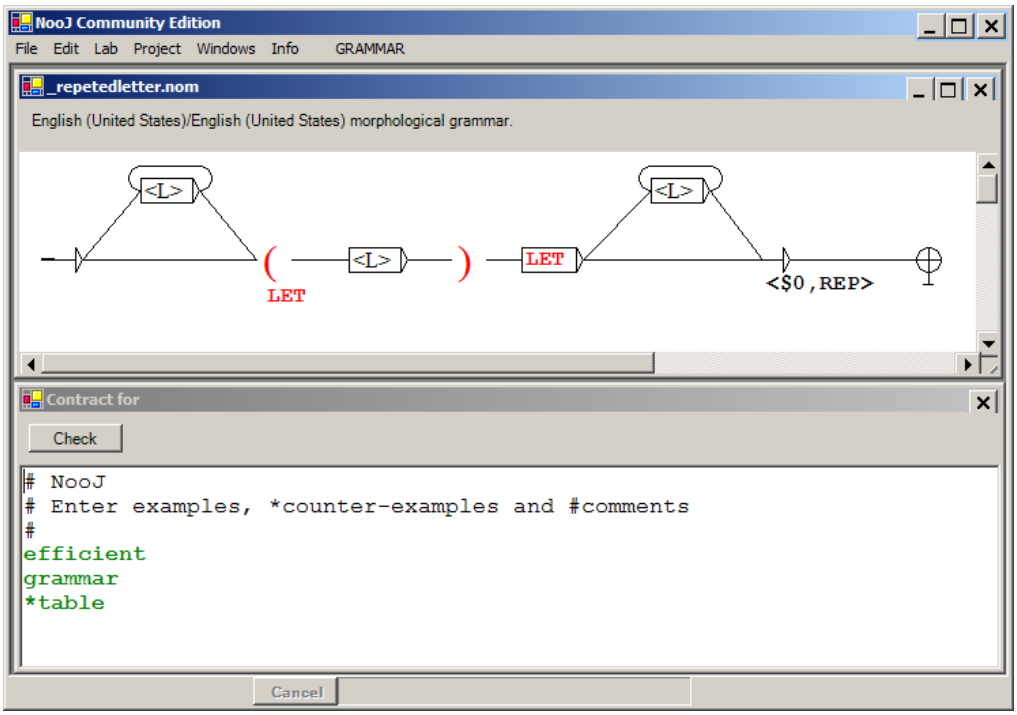


Figure 46. A grammar that recognizes wordforms with repeated letters

This grammar can be modified to recognize wordforms that include two or more letter repetitions (e.g. "Mississippi"), syllable repetitions (e.g. "barbarian"), letters that occur a certain number of times (e.g. "discussions"), wordforms that start and end the same way (e.g. "entertainment"), etc.

Lexical Constraints

The previous grammars can be used when one can describe the exact set of wordforms to be recognized, such as the roman numerals, or when the set of wordforms is "extremely" productive (i.e. with no exception), such as the proper names.

Indeed, the "generic" type of grammars that include symbols such as <L> has proven to be useful to quickly populate new dictionaries, and to automatically extract from large corpora lists of wordforms to be studied and described by lexicographers. For instance, the patterns "<L>* ize" and "re <L>*" can be applied to large corpora in order to extract a set of Verbs.

When using productive grammars, one usually gives them the lowest priority, so that wordforms already described in dictionaries are not being re-analyzed; for instance, in the previous grammar, we do not want the wordform "Autisme" (which is listed in the French dictionary) to be re-analyzed as a political word derived from the Proper name "Aut"!

In order to control the domain of application of a morphological grammar, it is important to produce **constraints** on various parts of the wordform that have to be checked against NooJ's lexical resources.

In NooJ, these constraints are parts of the output produced by the morphological grammar, and are also written between angles ("<" and ">"). Note that these are the same constraints that are used in NooJ dictionaries and by the NooJ query and syntactic parsers. For instance, consider the following grammar, similar to the "naïve" grammar used above:

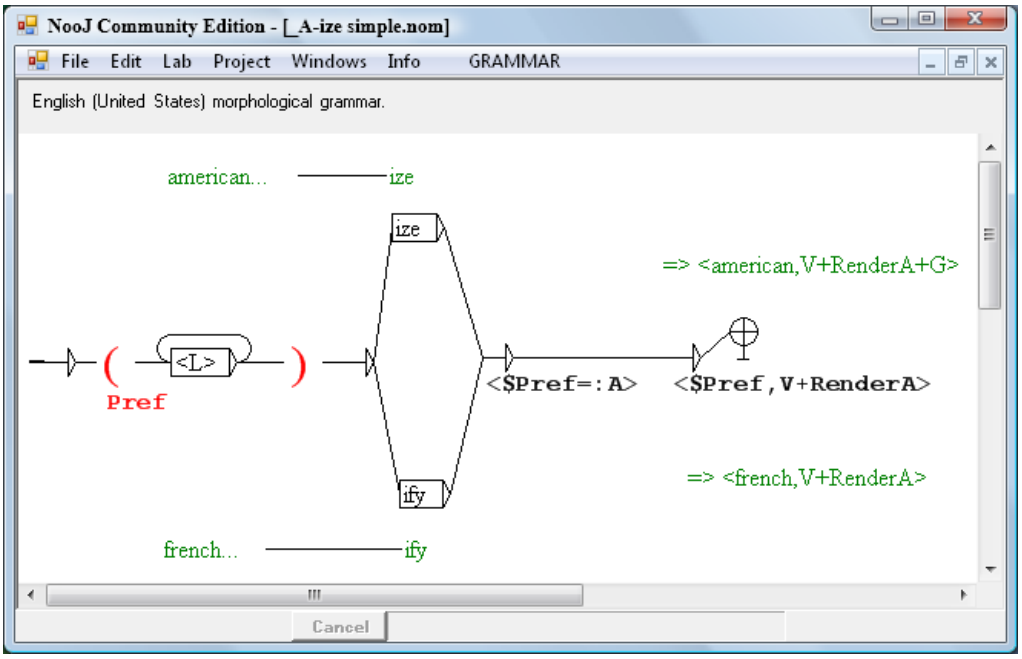


Figure 47. Adding a lexical constraint

Just like the previous grammar, this grammar recognizes wordforms such as "americanize" and also "size". However, for the wordform "americanize", the constraint **<$Pref=:A>** gets rewritten as **<american=:A>**; this constraint is validated by a dictionary lookup, i.e. NooJ checks that *american* is indeed listed as an adjective. On the other hand, for the wordform "size", the constraint **<s=:A>** does not check because there is no lexical entry "s" in NooJ's dictionary that is associated with the category "A". Therefore, only the first analysis is produced.

Lexical constraints can be as complex as necessary; they constitute an important feature of NooJ because they give the morphological module access to the precision

of any linguistic information stored in any NooJ dictionary. For instance, one can check all the verbs of a dictionary that derive to an "-able" adjective, and associate them with the morpho-syntactic feature "**+able**", as in: the following dictionary:

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

Then, in the following morphological grammar:

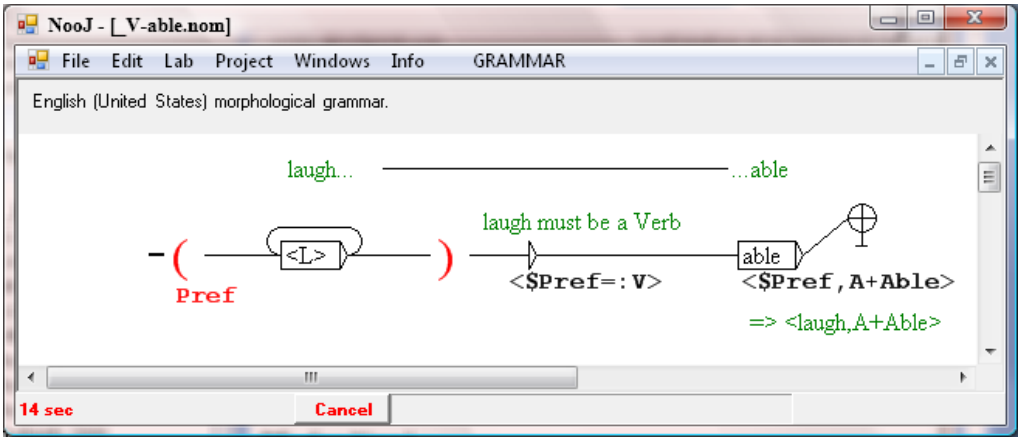


Figure 48. Derivation with a complex lexical constraint

the lexical constraint **<$Pref=:V+able>** ensures that derivations are performed only for verbs that are associated with the feature "**+able**". The resulting tag produces an Adjective, but the lemma is the initial Verb. For instance, the previous grammar produces the same analysis for the wordform "showable" as the one produced by the following lexical entry:

```
showable,show,A+Able
```

The wordforms "showable" and "laughable" are tagged only because the lexical entries "show" and "laugh" are associated with the category "**V**" and the feature "**+able**" in the dictionary.

On the other hand, the wordform "table" would not be recognized because the lexical constraint **<t=:V+able>** does not check: "t" is not a Verb. Similarly, the wordforms "sleepable" and "smileable" would not be recognized because the lexical entries "sleep" and "smile" are not associated with the feature "+able".

Furthermore, lexical constraints can (and should) be used to limit derivations to specific classes of words, e.g. only transitive verbs, only human nouns, or adjectives that belong to a certain distributional class. For instance, consider the following grammar:

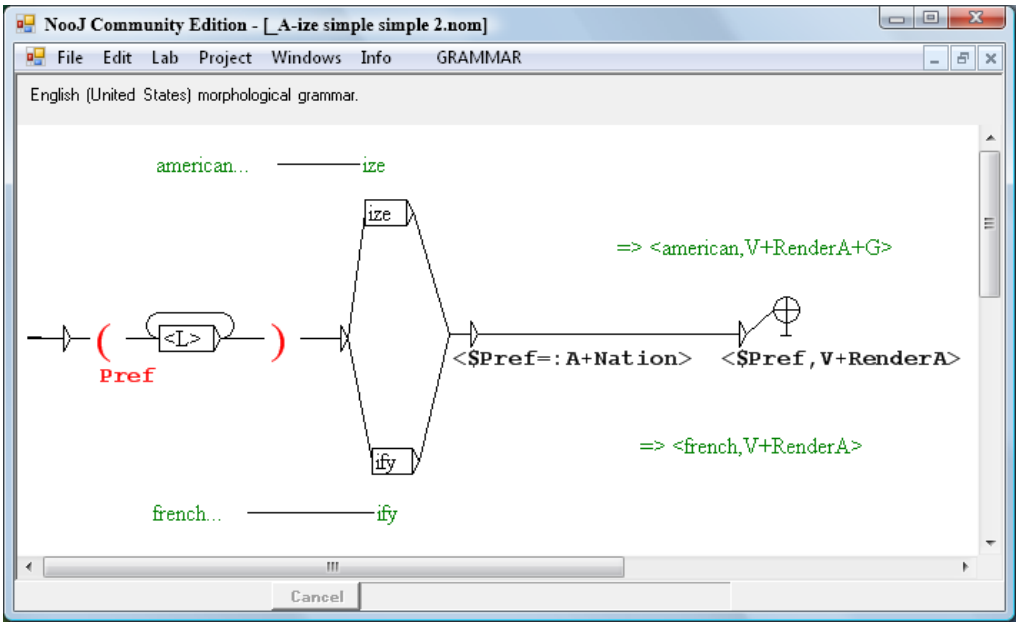


Figure 49. derivation with a complex constraint

It performs derivations only on Adjectives that are associated with the code "+Nation", such as "american" and "french", but would not apply to other types of adjectives, such as "big" or "expensive".

Negation

Just like in NooJ's query symbol, a constraint can contain a number of negative features, such as in:

<$Pref=:N-hum-plural>

(**$Pref** must be a Noun, not human and not plural). In the same manner, the right member of the constraint can be negated, such as in:

<$Pref=:!N>

(**$Pref** must not be a Noun). Finally, the constraint can include a global negation, such as in:

<!$Pref=:N+Hum>

(**$Pref** must not be a human noun). Note that the global negation is equivalent to the right-side negation.

Complex tokenizations

A grammar can produce more than one lexical constraint, so that each component of the wordform is associated with a specific constraint.

Being able to tokenize a wordform into a series of linguistic units is essential for Asian, Germanic and Semitic languages. For instance, the German wordform:

Schiffahrtsgesellschaft

should be associated with a sequence of three annotations such as:

```
<Schiff,N> <fahren,V+PP> <gesellschaft,N>
```

In this case, it is essential to make sure that each component of the whole wordform is indeed a valid lexical entry. Lexical constraints allow us to enforce that. In NooJ, this tokenization can be performed by the following specific graph:

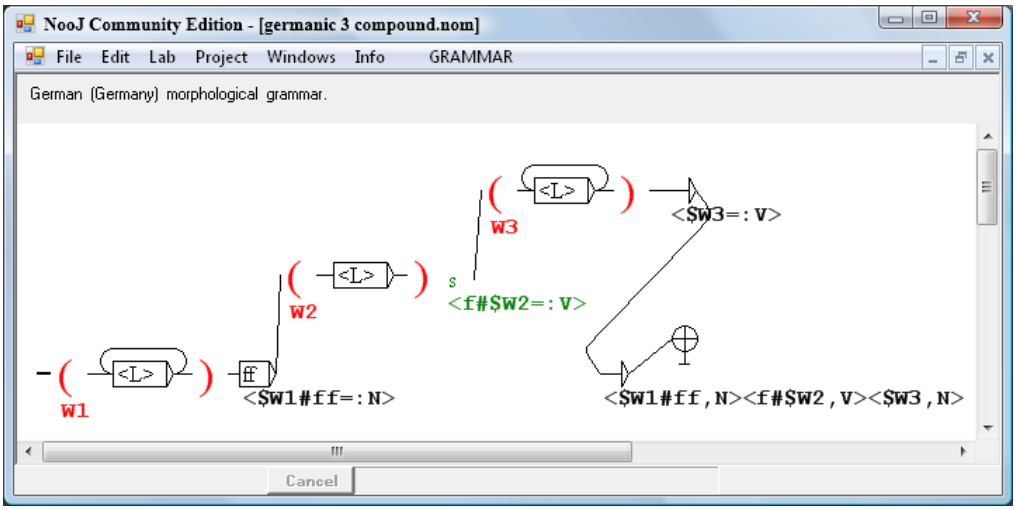


Figure 50. A complex tokenization in German

Notice that the third (missing) "f" between "Schiff" and "fahren" has to be re-introduced in the resulting tags, and how the extra "s" between "fahrt" and "gesellschaft" is deleted.

Variables, as well as lexical constraints, can be used in embedded graphs, as well as in loops. In this latter case, more than one occurrence of a given variable can be used along the path, with more than one value. In order to make sure that the correct value is used in the corresponding lexical constraint, make sure that each lexical constraint is inside the same loop as the variable, and in its immediate right context. For instance, consider the following grammar:

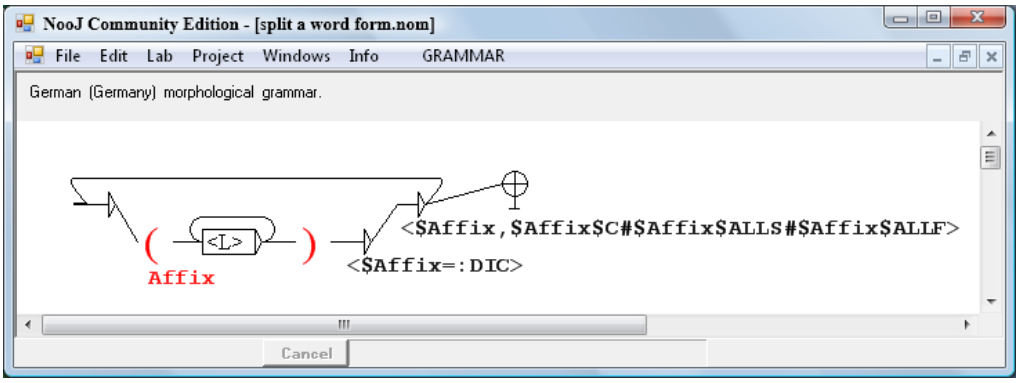


Figure 51. Lexical Constraints in a loop

If this grammar matches the complex wordform "underworkmen", the variable **$Affix** will be given three values successively: "under", "work" and "men".

Each occurrence of the variable **$Affix** is followed immediately by a lexical constraint that uses its current value: the variable **$Affix** is first set to "under", then the corresponding lexical constraint is **<under=:ALU>** (**ALU** matches any text unit described by a dictionary or a morphological grammar); then the variable is set to "work" and the following lexical constraint is **<work=:ALU>**; then the variable is set to "men" and the lexical constraint is set to **<men=:ALU>**. Each lexical constraint is followed by an annotation, and the full analysis of the wordform produces a sequence of three tokens:**<under,PREP><work,N><man,N+p>**.

Transfer of features and properties

NooJ enforces lexical constraints to affixes of the wordform 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 wordform "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), "+PRT" (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". NooJ 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 grammar; $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 and Semantic features
$1F = 1st constraint, in**F**lectional features

Now consider the following grammar:

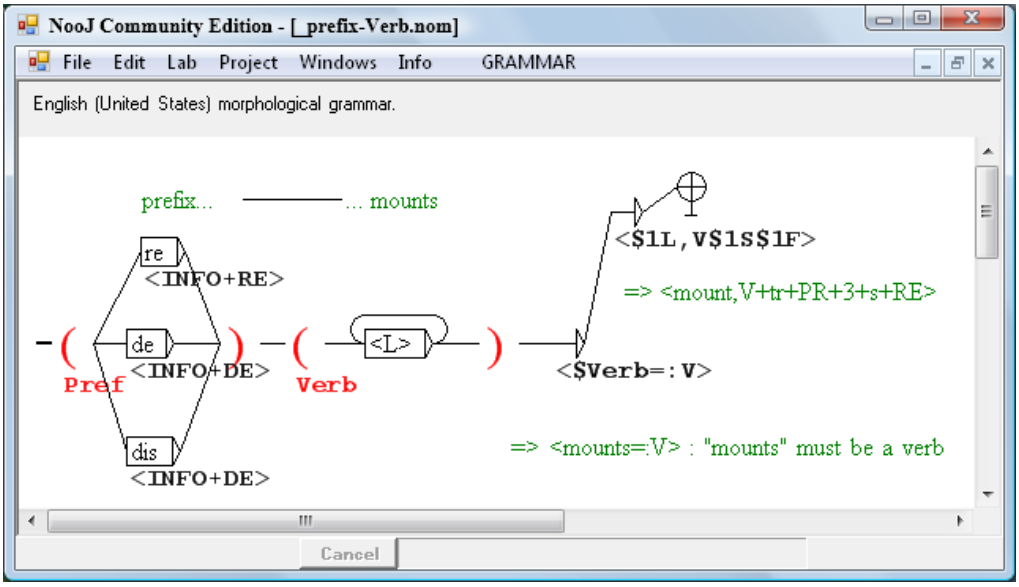


Figure 52. prefixes and verbs

It recognizes the wordform "dismounts" if **<$Verb=:V>** i.e. if the wordform "mounts" is associated with the category "**V**" (Verb). If this constraint checks, the grammar produces the resulting tag:

```
<$1L,V$1S$1F>
```

In this tag, the variable **$1L** stores the lemma of "mounts", i.e. "mount"; **$1S** stores the syntactic and semantic features for "mounts", i.e. "+tr" (transitive), and **$1F** stores the inflectional information for "mounts", i.e. "+PR+3+s" (PResent, third person, singular). As a result, the wordform "dismounts" is annotated as:

```
<mount,V+t+PR+3+s+DE>
```

Recursive constraints

The previous lexical constraints were enforced simply by looking up the selected dictionaries. It is also possible to write constraints that are enforced recursively, by looking up all lexical and morphological resources, including the current one. For instance, in the following grammar, the lexical constraint **<$Verb=:V>** checks recursively that the suffix of the current wordform is a Verb.

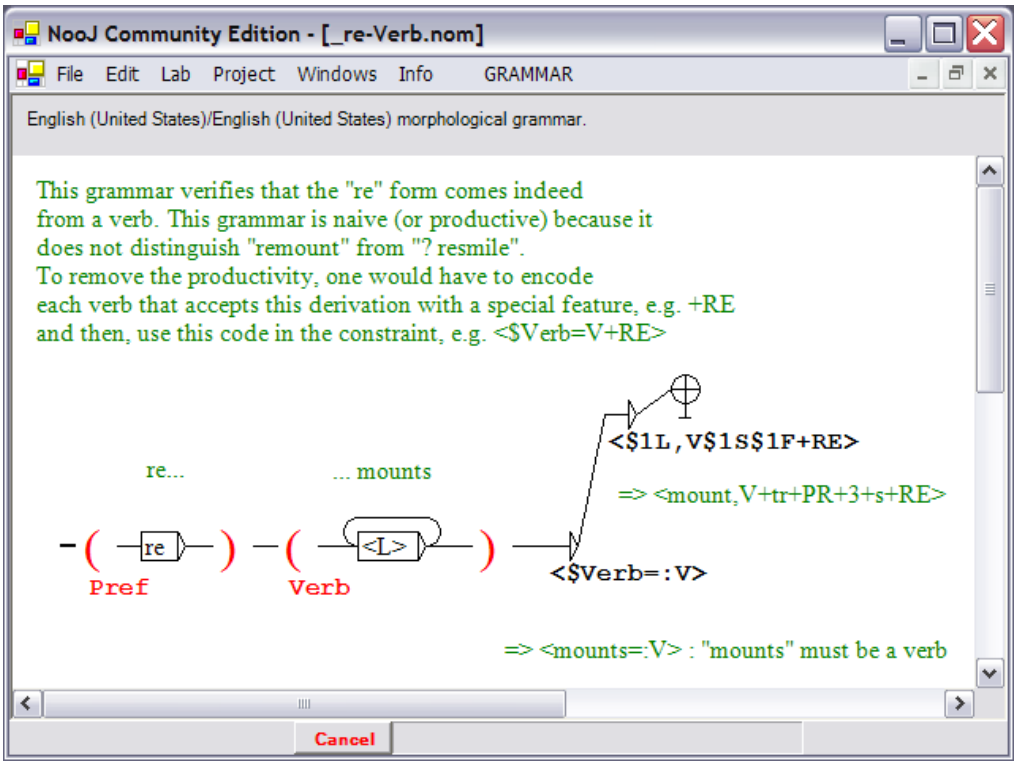


Figure 53. Recursive lexical constraint

When given the wordform "reremounts", the grammar produces the lexical constraint **<remounts=:V>**, that triggers the morphological analysis of the wordform "remounts". This wordform is then analyzed by the same grammar, that produces the lexical constraint **<mounts=:V>**, that checks OK thanks to a dictionary lookup. The final resulting tag is then:

```
<mount,V+tr+PR+3+s+RE+RE>
```

Notice that the feature **+RE** is produced twice.

Recursivity is important in Morphology because it allows linguists to describe each prefixation and suffixation independantly. For instance, a wordform such as "redismountable" can be recognized by independent grammar checks: the "**V-Able**" grammar produces the constraint **<redismount=:V>**, then the "**re-V**" grammar produces the constraint **<dismount=:V>**, then the "**de-V**" grammar producing the constraint **<mount=:V>**, then a final dictionary lookup checks OK.

Agreement Constraints

Lexical Constraints allow linguists to perform relatively simple checks on affixes: NooJ checks that a certain sequence of letters corresponds to a linguistic unit with such a property.

NooJ's morphological engine has two other operators that can be used to check the equality (or the inequality) of two properties: the equality operator "=" and the inequality operator "!=". For instance, the following agreement constraint:

<NomNumber=AdjNumber>

checks that the value of property "Number" of the affix stored in variable **$Nom** is equal to the value of property "Number" of the affix stored in variable **$Adj**. Conversely, the following agreement constraint:

<NomNumber!=AdjNumber>

checks that they differ.

Note that agreement constraints can be used to simulate lexical constraints. For instance, the two following constraints are equivalent if the affix stored in variable $Nom is a noun:

<NomNumber="plural">, <$Nom=:N+plural>

The special tag <INFO>

In various cases, some piece of the linguistic information that needs to be attached to the Atomic Linguistic Unit is produced along the way, not necessarily in the final order we wish to write the resulting annotation. In that case, we can use the special tag **<INFO>**, associated to the features and properties to be concatenated at the end of the annotation.

For instance, consider the following grammar:

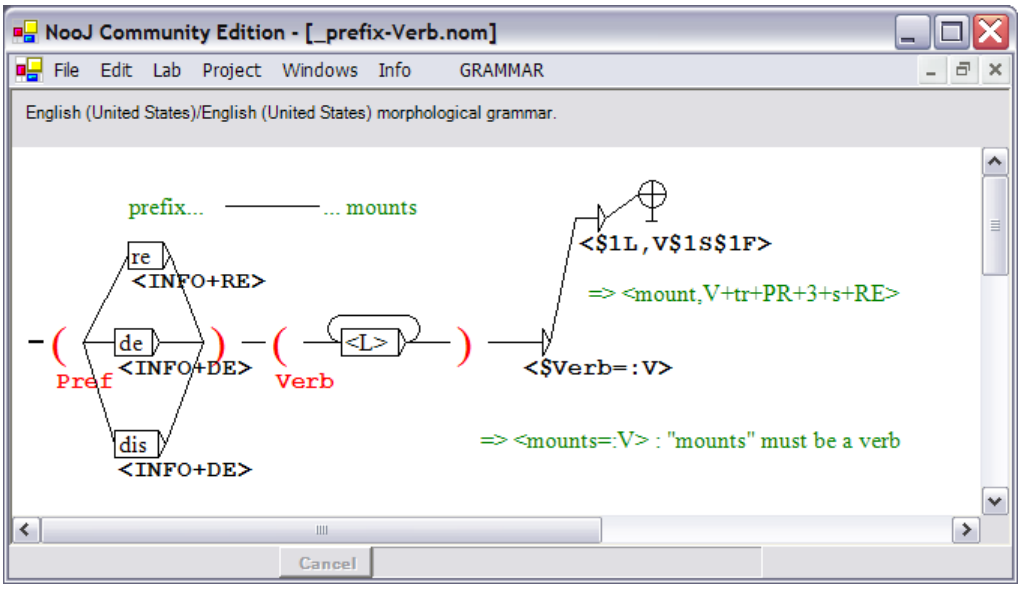


Figure 54. Use of the special tag <INFO>

It recognizes the wordforms "remount" as well as "dismounted". The features "+RE" and "+DE" are produced accordingly, but they will be concatenated at the end of the resulting annotations:

```
<mount,V+tr+INF+RE>
<mount,V+tr+PRT+DE>
```

Inflectional or Derivational analysis with morphological grammars

In Chapter 8, we described NooJ's inflectional and derivational engine. It is possible to simulate it by using morphological grammars.

In this case, we would use a dictionary with no inflectional or derivational information (i.e. no **+FLX** or **+DRV** features), and we would use instead lexical constraints linked to special features in the dictionary. For instance, here would be such a dictionary:

```
abandon,V+Conj3
help,V+Conj3
mount,V+Conj3
```

The feature "**+Conj3**" would then be used in a lexical constraint produced by a grammar such as the graph below.

This graph uses the lexical constraint **<$R=:V+Conj3>** together with the dictionary above in order to enforce that only Verbs listed in the dictionary, and associated with the conjugation paradigm **+Conj3**, will be recognized.

For instance, the graph recognizes the wordform "helps" (through the path at the bottom): variable **$R** stores the prefix "help"; NooJ looks up "help", and verifies that it exists indeed as a lexical entry associated with the information "V+Conj3". Then, the grammar produces the resulting analysis:

```
<helps,help,V+Conj3+PR+3+s>
```

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".

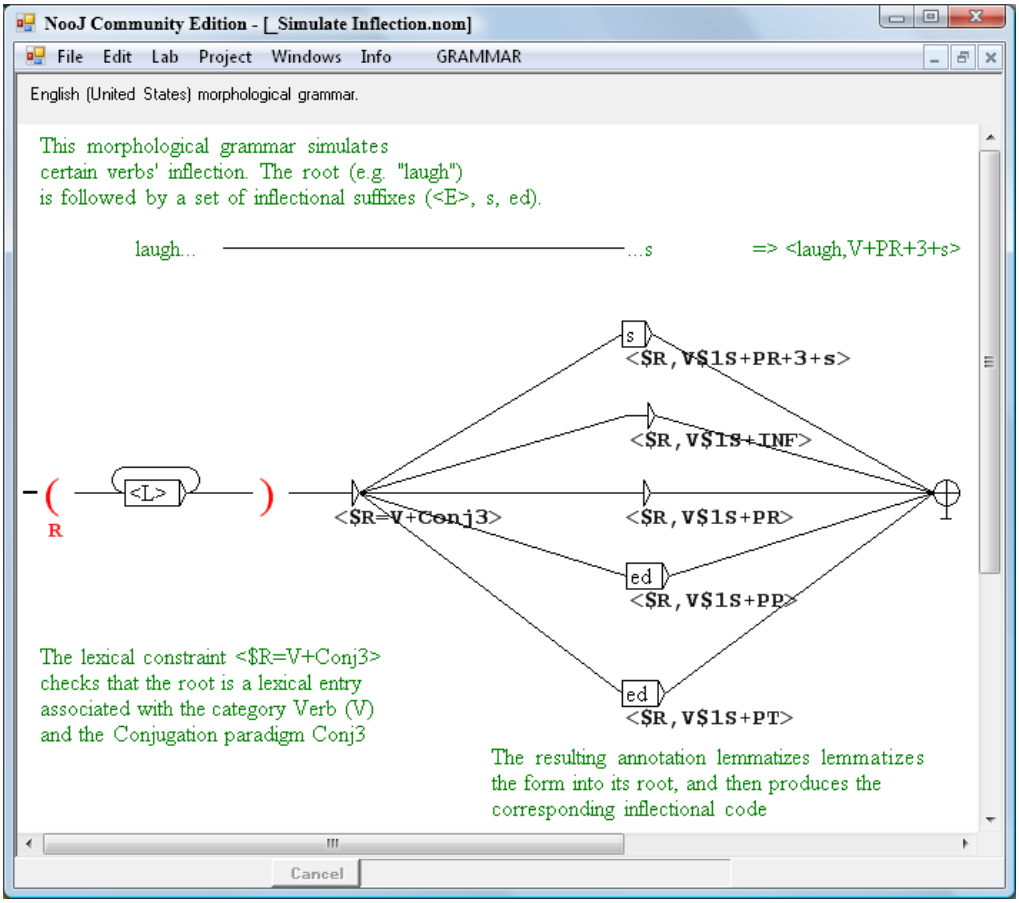


Figure 55. A morphological grammar that processes conjugation

12. Lexical Parsing

NooJ's dictionaries and morphological resources can be selected to be used by NooJ's lexical parser. In order to select a lexical resource, open the window **Info > Preferences > Lexical Analysis** (make sure that the current language is properly selected), and then check its name, in the upper zone of the window if it is a dictionary, or in the lower zone if it is a morphological grammar:

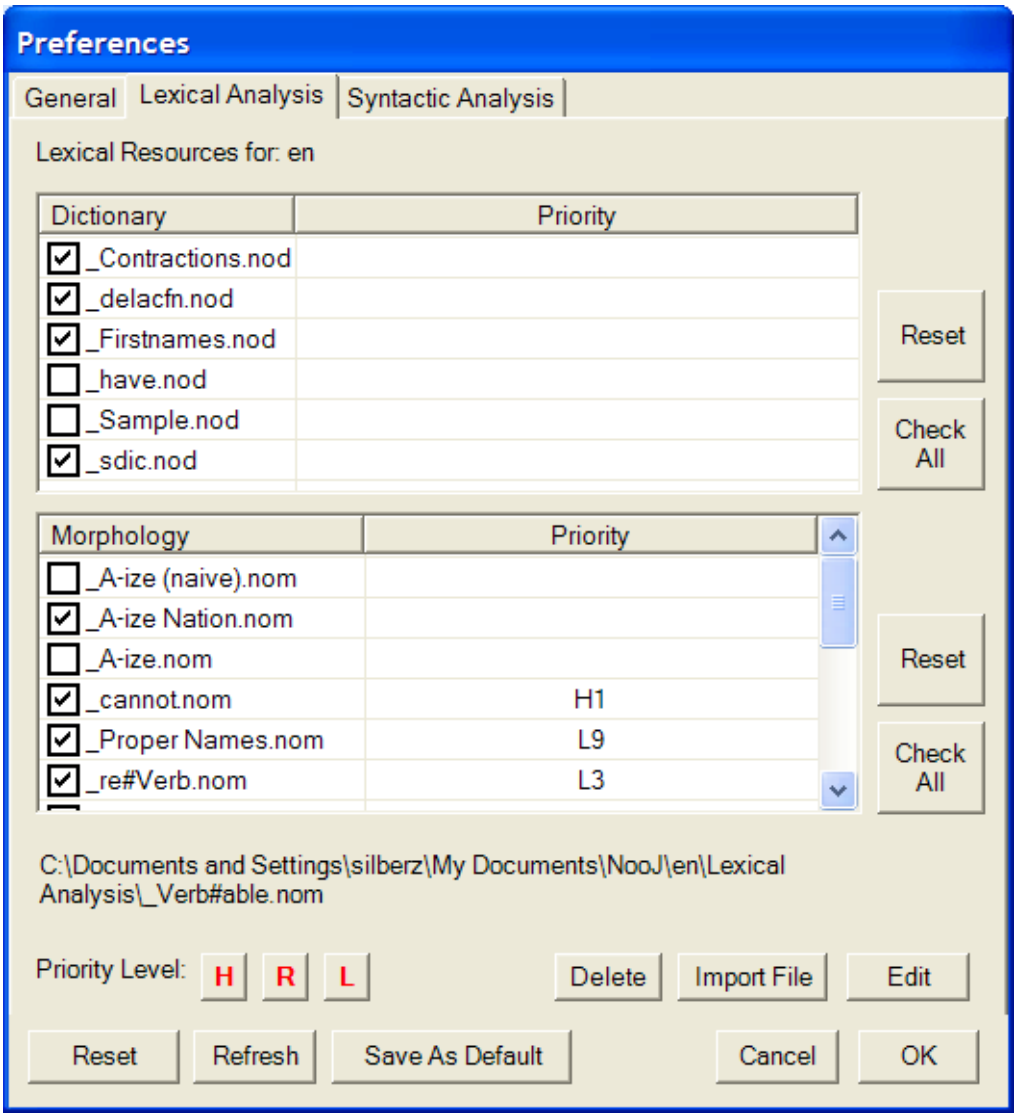


Figure 56. Select linguistic resources for the lexical parser

Users can select any number of dictionaries and morphological grammars so that NooJ's lexical parser applies them to texts every time the user performs a linguistic analysis of a text or a corpus.

To apply all selected lexical resources to the current text or corpus, use **TEXT > Linguistic Analysis**, or **CORPUS > Linguistic Analysis**.

It is easy to add a lexical resource to NooJ's pool of lexical resources:

-- in case of a morphological grammar (a file with extension ".nom"), just store it in the **Lexical Analysis** folder for the corresponding language; typically, on a Windows PC, the folder will look like:

```
My Documents\NooJ\en\Lexical Analysis
```

On a Mac, morphological grammars will be stored in the following folder:

-- in case of a dictionary (a file with extension ".dic"), compile it by using the Lab > Dictionary, and then store the resulting ".nod" file in the **Lexical Analysis** folder.

Priority levels

More than one lexical resource can be selected to be used by NooJ's lexical parser. If they have the same priority level, NooJ's lexical parser computes their union, and wordforms that are recognized by more than one lexical resource will typically produce more than one analysis.

One can also *hide* information thanks to a system of prioritization. Each lexical resource is associated with a priority level, that can be either "H" (High), "R" (Regular) or "L" (Low). When parsing a text,

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

(2) then if a wordform has not been recognized by the high-level lexical resources, NooJ applies the "regular priority" lexical resources;

(3) then, if both the consultation of "high priority" and "regular priority" lexical resources have not produced any result, NooJ applies the "low priority" lexical resources.

Furthermore, there are degrees in High and Low priority levels, so that "H9" has the highest "High Priority" level, "H1" has the lowest "High Priority" level, "L1" is the highest priority among "Low priority resources" and "L9" is the lowest of all resources.

This system allows users to hide or add linguistic information at will. For example, NooJ's English dictionary **sdic** usually has the "default" priority. It describes numerous usages that are not overly frequent, for example *and* = *Verb* in a technical text, e.g. *"we anded the two binary values*". For a less specific application, as when processing standard texts that are not technical in nature (literature or newspapers), in which these words would never appear, it is useful to create a smaller dictionary which has a higher priority than the **sdic** dictionary, for instance "**H1**", in which technical uses are not described, e.g. *and* is only described as a conjunction. In effect, This small dictionary will hide useless **sdic** entries and act as a filter, to filter out unwanted entries.

To give a high priority to a lexical resource, select the resource's name in the **Info > Preference > Lexical Analysis** panel, then click the button "H". To give a low priority to a lexical resource, select the resource's name then click the button "L". To give a regular priority to a lexical resource, select the resource's name then click the button "R".

The lexical resources associated with lower priority levels are applied when the application of the other lexical resources has failed. We generally use this mechanism to process unknown wordforms. For instance, in the last chapter, the grammar to recognize proper names is applied only to unknown wordforms: all simple forms not found in NooJ's dictionaries and which begin with a capital letter, are identified by this grammar. Similar grammars can be used to recognize productive morphological derivations such as *redestalinization*, *redestructurability*, etc.

Disambiguation using high-priority dictionaries

We can use the system of priorities to eliminate artificial ambiguities. For example, the following multi-word units occur frequently in texts:

as a matter of fact, as soon as possible, as far as I am concerned

They do not have non-autonomous constituents, therefore, NooJ will systematically suggest two analyses: multi-word unit (eg. The adverb *as a matter of fact*), or the sequence of simple words (eg. The conjunction *as* followed by the determiner *a*, followed by the noun *matter*, followed by the preposition *of*, followed by the noun *fact*). But in truth, these multi-word units are not ambiguous. To avoid producing artificial ambiguities, we store these words in "high-priority" dictionaries, i.e. in dictionaries which are associated with priority levels above NooJ's standard dictionary **sdic**.

Note that this mechanism can also be used to adapt NooJ to specific vocabularies of domain languages. For instance, if we know that in a specific corpus, the word "a" will always refer to the determiner (and never to the noun), we could enter the following lexical entry:

```
a,DET
```

in a small, high-level priority dictionary adapted to the corpus.

Text Annotation Structure

The application to a text (**TEXT > Linguistic Analysis**) of the lexical resources that are selected in **Info > Preferences > Lexical Analysis** builds the Text's Annotation Structure (**TAS**), in which all recognized Atomic Linguistic Units (ALUs), whether

affixes, simple words or multi-word units, are associated with one ore more annotations.

The text's annotation structure can be displayed via the check box "**Show Annotations**". Make sure to resize the lower part of the window in order to give it enough vertical space, so that all ambiguities are shown:

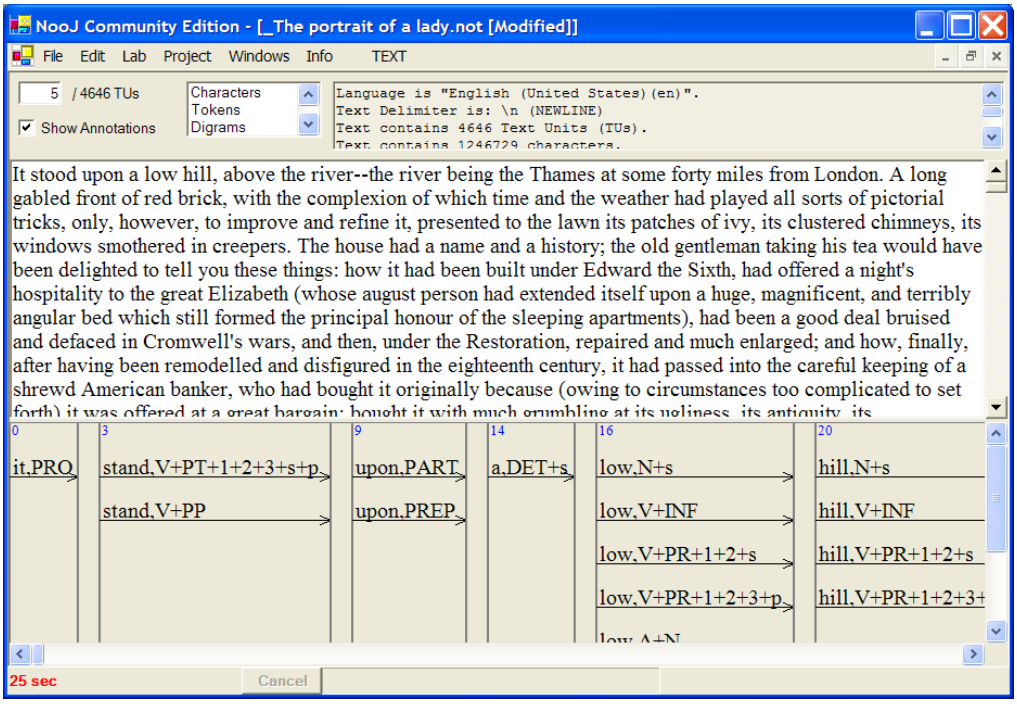


Figure 57. Text Annotation Structure

Annotations and Ambiguities

Note that ambiguities are numerous; at this stage, we have only added lexical and morphological annotations to the TAS; we will see later (in the chapter on Syntactic Analysis) how to remove annotations from the TAS.

Several types of ambiguities can occur:

-- one wordform corresponds to more than one lexical entry, in one or more dictionary. For instance, the wordform "being" is either a conjugated form of the lexical entry "be" (the verb *to be*), or the noun (*a being*).

-- one wordform can correspond to one or more lexical entries, and at the same time, to the result of a morphological analysis. For instance, the wordform "retreat" can be a verb form (e.g. *the army retreats*), or the concatenation of the repetition prefix "re" followed by a verb "treat" (e.g. *the computer retreats the query*).

-- one sequence of tokens can correspond to one or more multi-word units, and at the same time, to a sequence of simple words. For instance, the text "... round table ..."

can be associated with one annotation (the noun meaning "a meeting"), or with two annotations (the adjective, followed by a noun, in the case of a round piece of furniture).

Ambiguities can be managed via NooJ's lexical resources' priority system (see above) and with the **+UNAMB** feature (see below). For instance, if we want a list of technical terms such as "nuclear plant" or "personal computer" never to be parsed as ambiguous with the corresponding sequences of simple words (e.g. the adjective "nuclear" followed by the noun "plant"), we can give the technical terms' dictionary a higher priority (e.g. "H3") than NooJ's dictionary **sdic**. If a given lexical entry should be processed as is, and disable any other possible analyses, we add the feature "**+UNAMB**" to its information codes; for instance, to always process the word "adorable" as an adjective (rather than a verb form followed by the suffix "-able"), we enter the wordform with the code **+UNAMB** in a dictionary:

```
adorable,A+UNAMB
```

Exporting the list of annotations and unknowns

At the top of the window, in the results zone after "Characters", "Tokens" and "Digrams", you can double-click "Annotations" to get a list of all the annotations that are stored in the TAS in a dictionary format. This dictionary can in turn be edited and exported.

In the same manner, double-click "Unknowns" to get a list of all the tokens that have no associated annotation. The "Unknowns" dictionary can be edited, for instance to replace the category "UNKNOWN" with the valid one for each entry. The resulting dictionary can then be compiled (**Lab > Dictionary**), selected in the **Info > Preferences > Lexical Analysis** window, and then re-applied to the text (**Text > Linguistic Analysis**) in a few minutes.

Unknowns do not always correspond to errors in the text or in the dictionaries. For instance, the wordform "priori" is not a lexical entry, although the multi-word unit "a priori" can be listed as a compound adverb. In consequence, the adverb "a priori" will show up in the "Annotations" window, but the wordform "priori", which is not associated with any annotation, will show up in the "Unknowns" window.

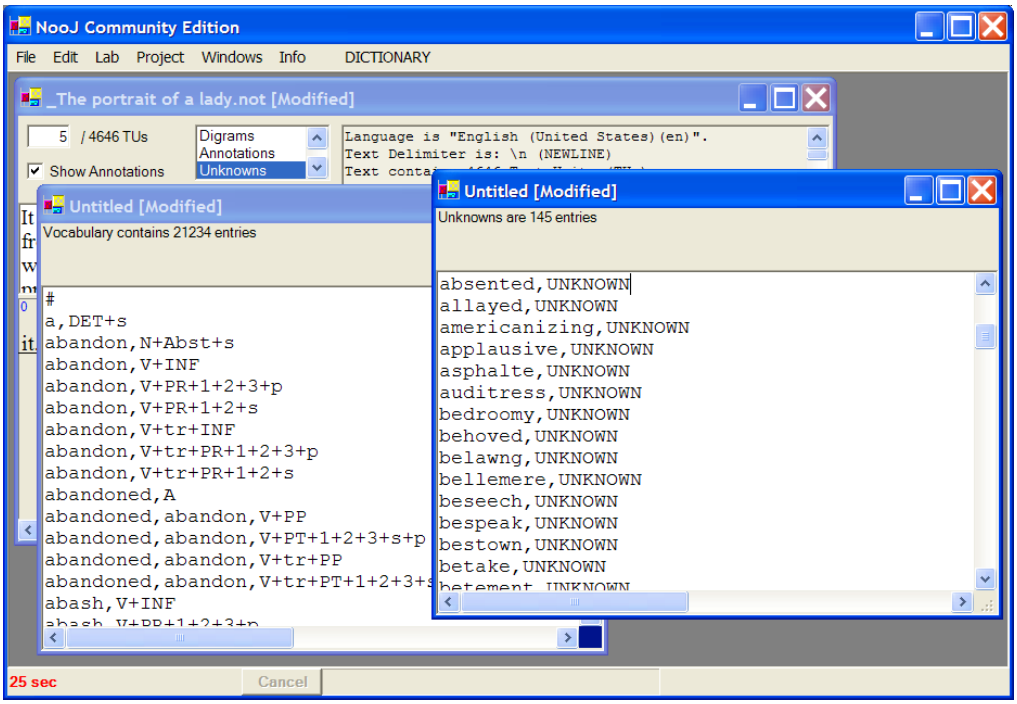


Figure 58. Export the text's annotations and unknowns as dictionaries

Special feature +UNAMB

+UNAMB stands for "Unambiguous". Inside one given lexical resource (dictionary or morphological grammar), there are cases where some solutions should take precedence over other ones.

For instance, consider the two following lexical entries:

```
United States of America,N
United States,N
```

These two entries could be used to recognize all occurrences of both sequences "United States of America" and "United States". However, consider the following text:

... The United States of America have declared ...

Looking up the previous dictionary will get *two* matches, because both lexical entries (including the shorter one *United States*) are indeed found in the text. In other words, the sequence "United States of America" will be analyzed either as one noun, or as a sequence of one noun ("United States") followed by "of", followed by "America".

This problem occurs very often, because most multi-word units are ambiguous, either with sub-sequences of smaller units, or with sequences of simple words. For instance, the term "nuclear plant" will be processed as ambiguous if the selected dictionaries contain the following three lexical entries:

```
nuclear,A
plant,N
nuclear plant,A
```

In order to solve these systematic ambiguities, we can add a **+UNAMB** feature ("Unambiguous") to the multi-word entries:

```
nuclear plant,N+UNAMB
United States of America,N+UNAMB
United States,N
```

When NooJ locates unambiguous lexical entries in the text, it gives priority to them, thus does not even apply other lexical resources to analyze sub-sequences of the matching sequence. In other words, NooJ will simply ignore the lexical entry "United States" when analyzing the text "United States of America"; in the same manner, it will ignore the lexical entries "nuclear" and "plant" when analyzing the text "nuclear plant".

However, we still have a problem: let's add to the previous dictionary the two following entries:

```
States,N
United,A
```

The text "United States of America" is still parsed as one unambiguous Atomic Linguistic Unit (because "United States of America" is marked as **+UNAMB**); however, the text "United States" will be parsed as ambiguous: either the noun "United States", or the sequence of two words "United" followed by "States". In order to get rid of the last ambiguity, we need to mark "United States" as +UNAMB as well. The final dictionary is then:

```
States,N
United,A
United States of America,N+UNAMB
United States,N+UNAMB
```

Now, if more than one unambiguous lexical entries, of varying lengths, are recognized at the same position, then only the longest entries will be taken into consideration by the system. In conclusion, both text sequences "United States of America" and "United States" will now be analyzed as unambiguous.

Note finally that it is possible to get ambiguities between lexical entries of the same length; in that case, the **+UNAMB** feature is used to filter out all other non-ambiguous entries. For instance, the following two lexical entries:

```
round table,N+UNAMB+"meeting"
round table,N+UNAMB+"knights of the..."
```

are both tagged **+UNAMB** in order to inhibit the analysis "*round* followed by *table*". However, both analyses "*meeting*" and "*knights of the...*" will be produced by NooJ's lexical parser.

Special feature +NW

+NW stands for "non-word". This feature can be added to lexical entries that are not supposed to be right outputs for NooJ's lexical parser.

Why would then one want to add a lexical entry to a dictionary, and then associate it with the **+NW** feature? For Romance languages and English, the tradition is to describe words by entering their lemmas in dictionaries (e.g. verbs are represented by their infinitive form). But we could also represent words by their stems or roots. For instance, consider the following dictionary entry for the verb "to have":

```
ha,V+NW
```

and then use the following morphological grammar to analyze all conjugated forms of "to have":

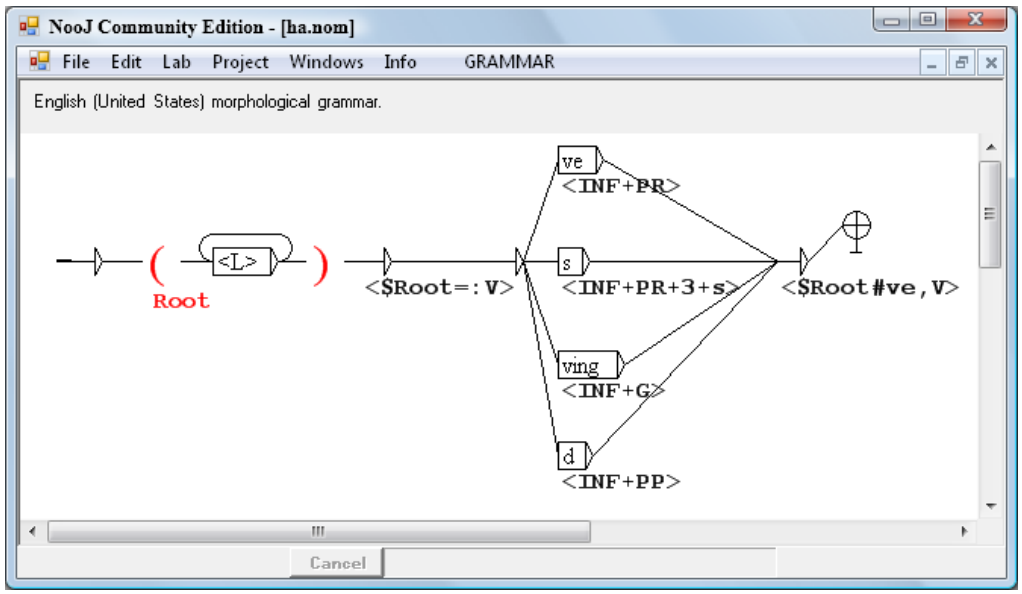


Figure 59. Morphological Analysis from a stem

This morphological grammar reconizes the wordforms *have*, *has*, *having* and *had*. Each of these forms will be computed from the root "ha", which needs to be listed as a verb in a dictionary, as required by the lexical constraint **<$Root=:V>**. But the wordform "ha" itself must not be recognized as a valid lexical entry from the dictionary. Hence, we associated the lexical entry "ha" with the feature **+NW** (non-word):

```
ha,V+NW
```

This functionality allows linguists to enter "non-words" in their dictionaries; these non-words are used by NooJ's morphological parser, but they will never be produced as plain annotations in a Text Annotation Structure. Note that NooJ's Hungarian module contains indeed a dictionary of "non-word" stems, which are associated with a series of morphological grammars.

Special category NW

The special category **NW** is used in a slightly different manner from the special feature **+NW**. It too must be associated to "non-words" or "artifacts", i.e. lexical entries that must not result in real annotations, and thus do not show up in the Text Annotation Structure. The difference between these two codes is that lexical entries associated with the **NW** category are still visible to NooJ's Syntactic Parser, whereas lexical entries associated with the feature **+NW** are not.

In consequence, the **NW** category can be used to describe components of expressions that can be parsed by NooJ's syntactic parser, but that do not occur as autonomous lexical entries. For instance, consider the following entries:

```
fortiori,NW+LATIN
posteriori,NW+LATIN
priori,NW+LATIN
```

These lexical entries will never produce lexical annotations, and are not displayed as annotations in the Text Annotation Structure. However, they can still be managed by NooJ's syntactic parser. For instance, the following syntactic grammar:

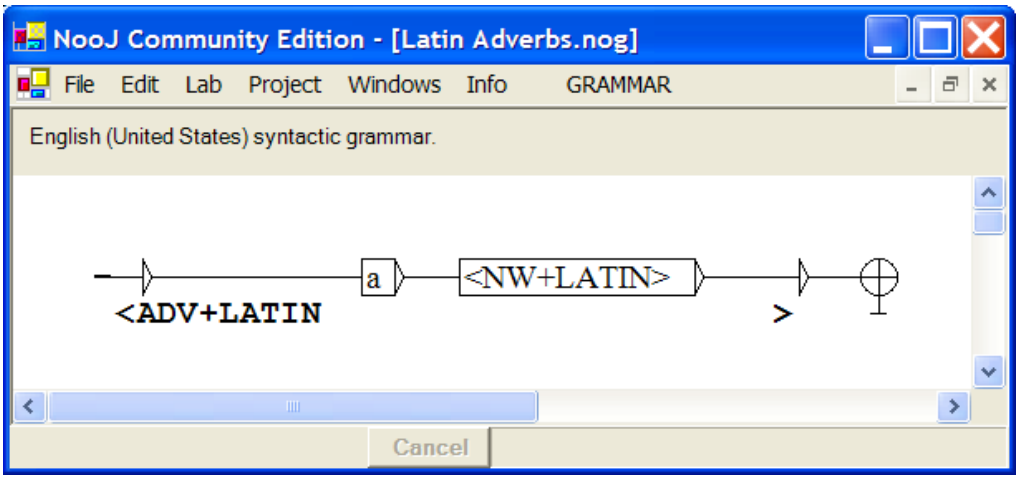


Figure 60. Latin Adverbs

will reconize the sequence “a priori” (“a” followed by a non word <**NW>**), and then will annotate it as an adverb (**<ADV+LATIN>**), even though the component “priori” will not be annotated.

SYNTACTIC ANALYSIS

This section presents syntactic grammars, i.e. grammars that process sequences of tokens (as opposed to the morphological grammars that process sequences of letters in tokens). In NooJ, syntactic grammars are organized sets of graphs, and are called **local grammars**. Chapter 13 presents local grammars that are used to recognize and annotate multi-word units and semi-frozen expressions, build complex embedded annotations, as well as delete annotations to disambiguate words. Chapter 14 shows how to formalize frozen expressions (i.e. discontinuous ALUs). Chapter 15 describes NooJ's syntactic parser's behavior in special cases, such as when applying ambiguous grammars. Chapter 16 presents more powerful syntactic tools, such as a Context-Free parser that can compute sentences' structure, and NooJ's Enhanced Grammars that can perform semantic analyses (i.e. to produce abstract representations of texts), transformations on texts (to generate paraphrases) and translations.

13. Syntactic grammars

In Chap. 11, we saw that certain parts of the dictionary can be processed with morphological grammars (see grammars **tsar** and **Roman numerals**). These grammars were applied by NooJ's morphological engine to simple wordforms (sequences of letters between delimiters); hence, they could not process multi-word units or frozen expressions.

The syntactic grammars we see now are also used to represent **ALU**s (Atomic Linguistic Units), but these ALUs can be multi-word units or frozen expressions.

Local grammars

Numeric Determiners

We want to identify numeric determiners written out in text form (e.g. *two thousand three hundred fifty two*). Start by creating a new syntactic grammar (**File > New Grammar**, choose the two languages en / en, then click "**Create a syntactic grammar**"), and name it **Dnum** (do not delete the default grammar **_Dnum**).

In the main graph of the new grammar, include links to three embedded graphs: "Dnum 2-99", "Dnum 100-999" and "Dnum 1000-999999", as seen in the following graph:

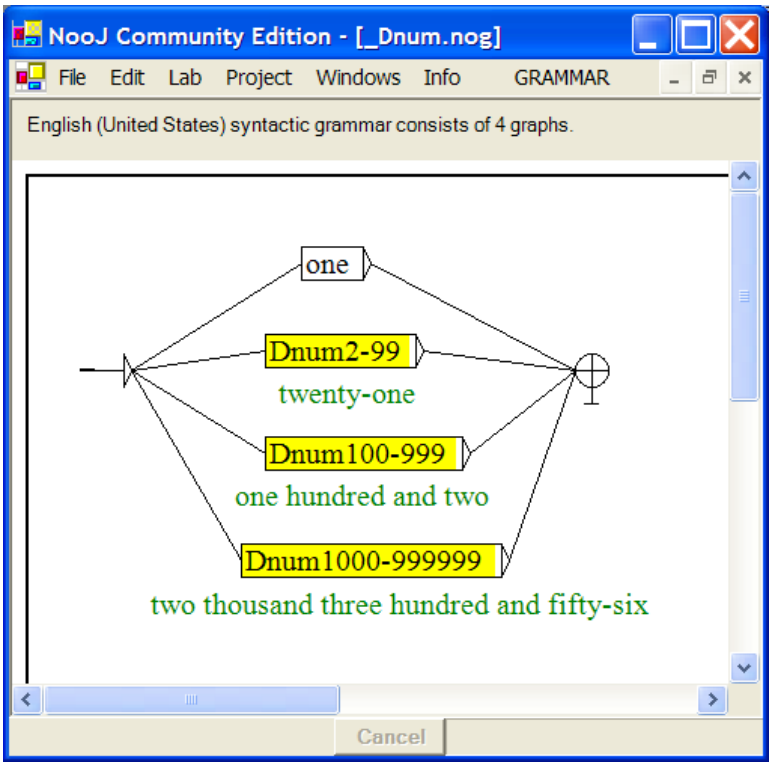


Figure 61. Numeric determiners from 1 to 99

To enter a reference to a graph in a node, write the graph's name prefixed with the colon special character ":". The node's background color should change to yellow to indicate that it is an **auxiliary node**.

At this point, the three graphs do not exist yet in the **Dnum** grammar. We are going to create the first one **Dnum2-99** by alt-clicking its reference. NooJ knows that the graph **Dnum2-99** does not exist, and asks if you want to create the graph:

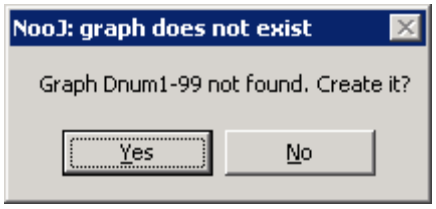


Figure 62. Create a new graph in a grammar

Answer "Yes". NooJ should now display an empty graph. Edit it and construct a graph similar to the following one:

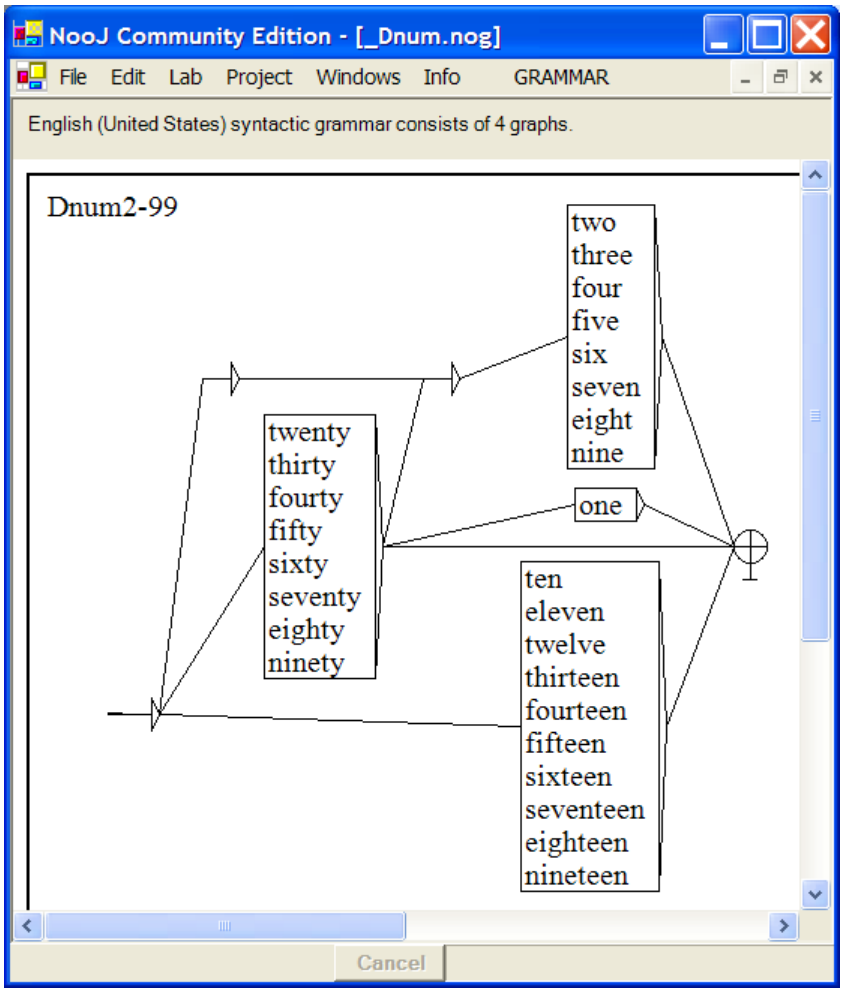


Figure 63. Numeric determiners from 1 to 99

Go back up to the Parent node (press the "U" key, or display the grammar's structure and click the **Main** graph) and then alt-click the node for **Dnum100-999** to create it.

To navigate between different graphs in one grammar, use:
-- the "U" key (UP) to display the current graph's Parent
-- the "D" key (DOWN) to display the current graph's first Child
-- the "N" key (NEXT) to display the current graph's next Sibling
-- the "P" key (PREVIOUS) to display the current graph's previous sibling
You can also display the grammar's structure Grammar > Show Structure, and navigate to any graph by clicking its name.

Enter the graph **Dnum100-999**, then edit it to get a graph similar to the following:

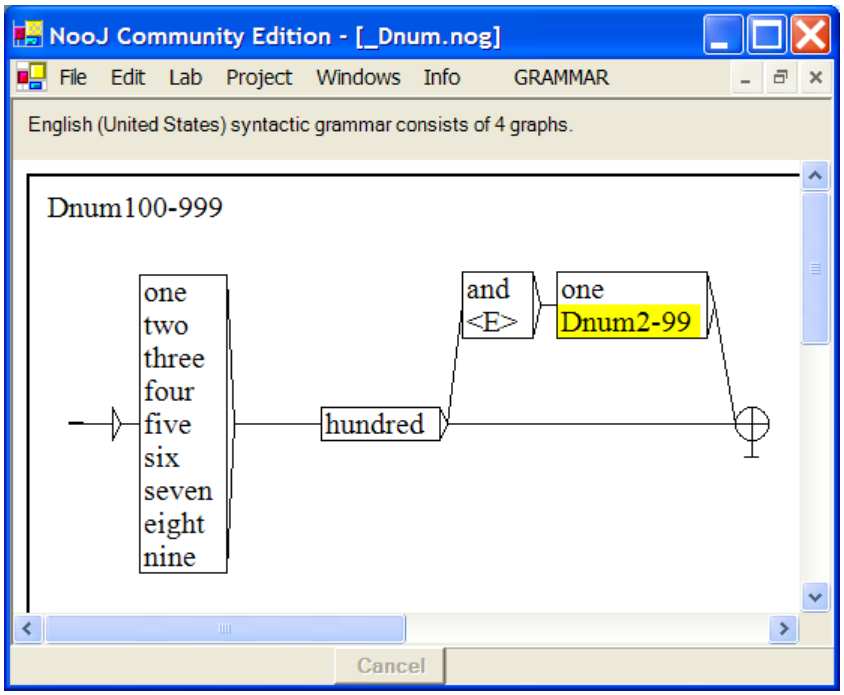


Figure 64. Numeric determiners from 100 to 999

Notice the yellow line "Dnum2-99" inside the graph "Dnum100-999" : it refers to the graph of the same name. If "Dnum2-99" is not displayed in yellow, remember to prefix its name with the colon character ":"; In this case we've entered the label "**:Dnum2-99**".

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

Continue the formalization of numeric determinants by constructing the graph "**Dnum1000-999999**" which will represent the numbers 1,000 to 999,999:

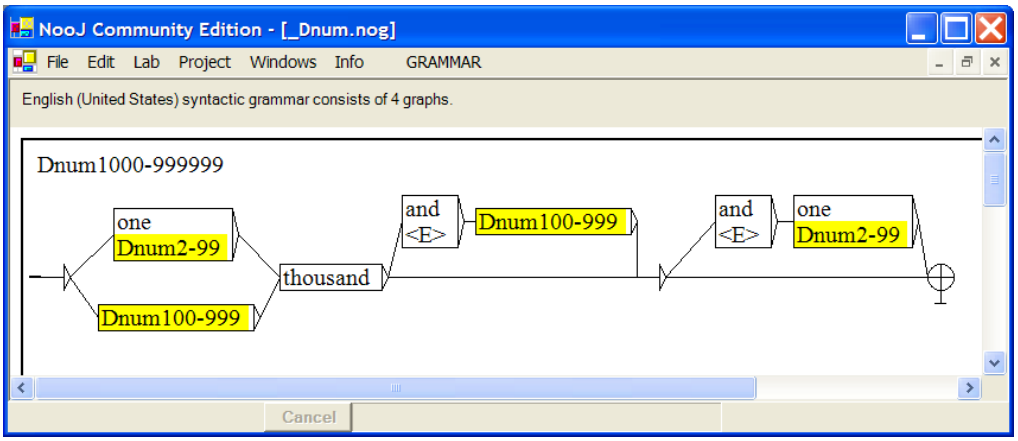


Figure 65. Numeric determiners from 1,000 to 999,999

Exercise: enhance the linguistic description to represent numeric determiners such as *thirteen hundred*, *hundreds of*, *thousands of*, *two millions thirty thousand and forty-three*, *eight billion*, etc.

Do not forget to save your grammar. At this point, we have created a grammar that can be applied to texts in order to build a concordance of numeric determiners. For instance, load the text "_en Portrait Of A Lady" (**File > Open > Text**), then apply the previous grammar to the text (Text > Locate), then:

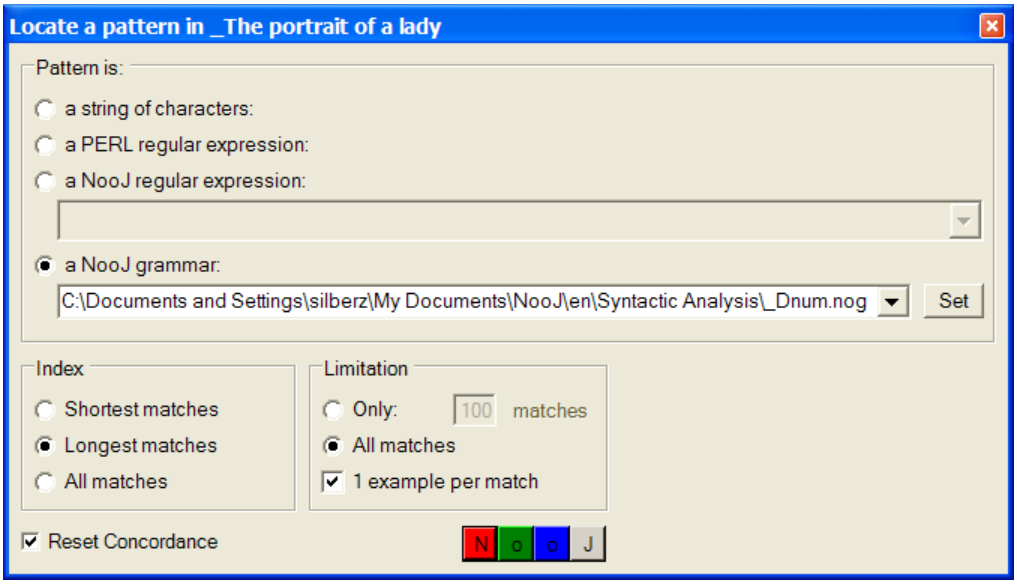


Figure 66. Locate numeric determiners in a text

With the same options checked (all matches; only one example per match), you should get a concordance with 33 entries:

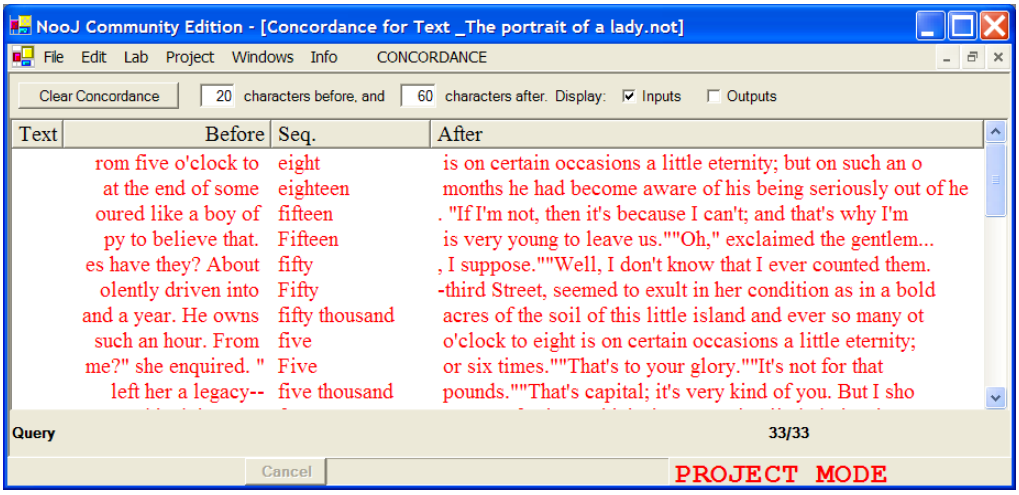


Figure 67. numeric determiners in the text *The portrait of a lady*

We are going to enhance this grammar so that instead of merely recognizing numeric determiners, NooJ will tag them in texts, just as if they had been entered in a dictionary.

Now, we want to annotate the matching sequences with the category "**DET**" (already used to tag other determiners such as "the"). In order to do that, we go back to the main graph (press the "U" key, or click the **Main** graph in the grammar's structure's window), then edit the initial node and enter the new label "<E>/<DET", add a new node before the terminal node, and enter its label "<E>/>". You should get a graph similar to the following:

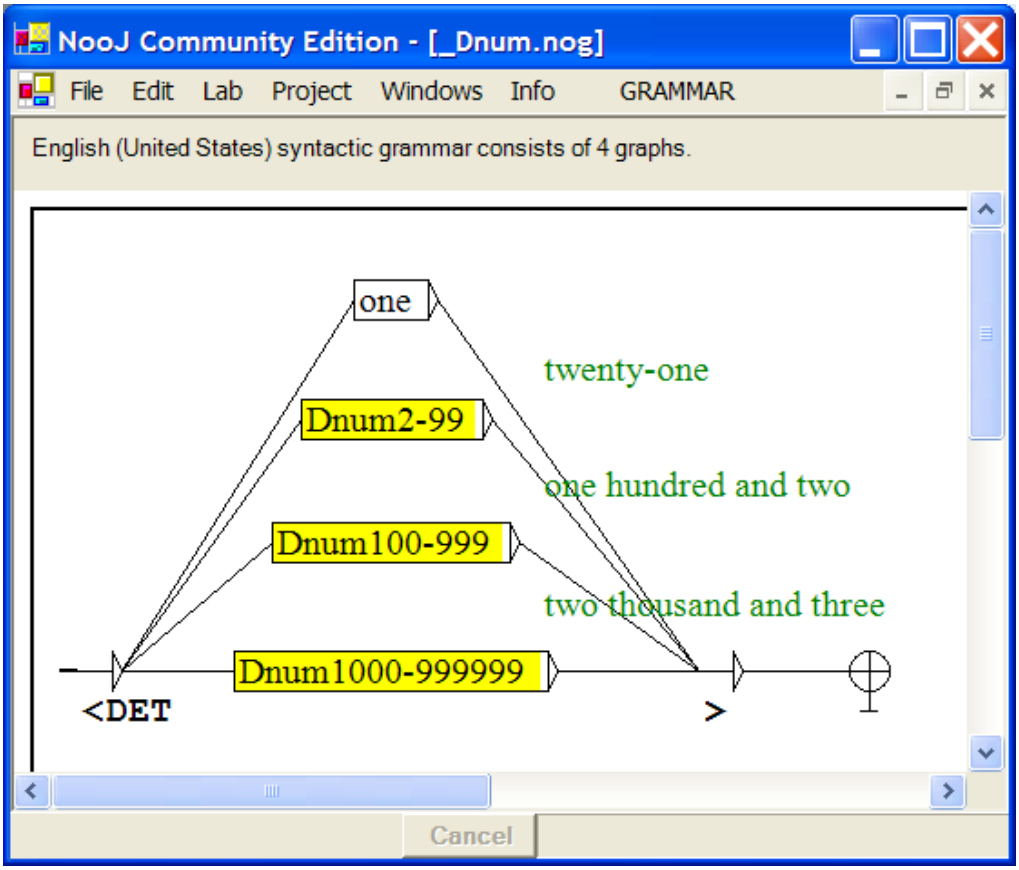


Figure 68. Annotating the numeric determiners

All the sequences that are recognized by this grammar will now be analyzed as "**DET**". Each annotation starts where the "<" character is located (in this case, at the beginning of the expression), and ends where the ">" character is located (in this case: at the end of the sequence).

We want to add more information than just the category: edit the graph to add the "+Num" property (numeric expression), the "one" label to add the "+s" (singular) property, then connect all other plural determiners to a new node labeled "<E>/+p" (plural). Connect the nodes so that you get a graph similar to the following:

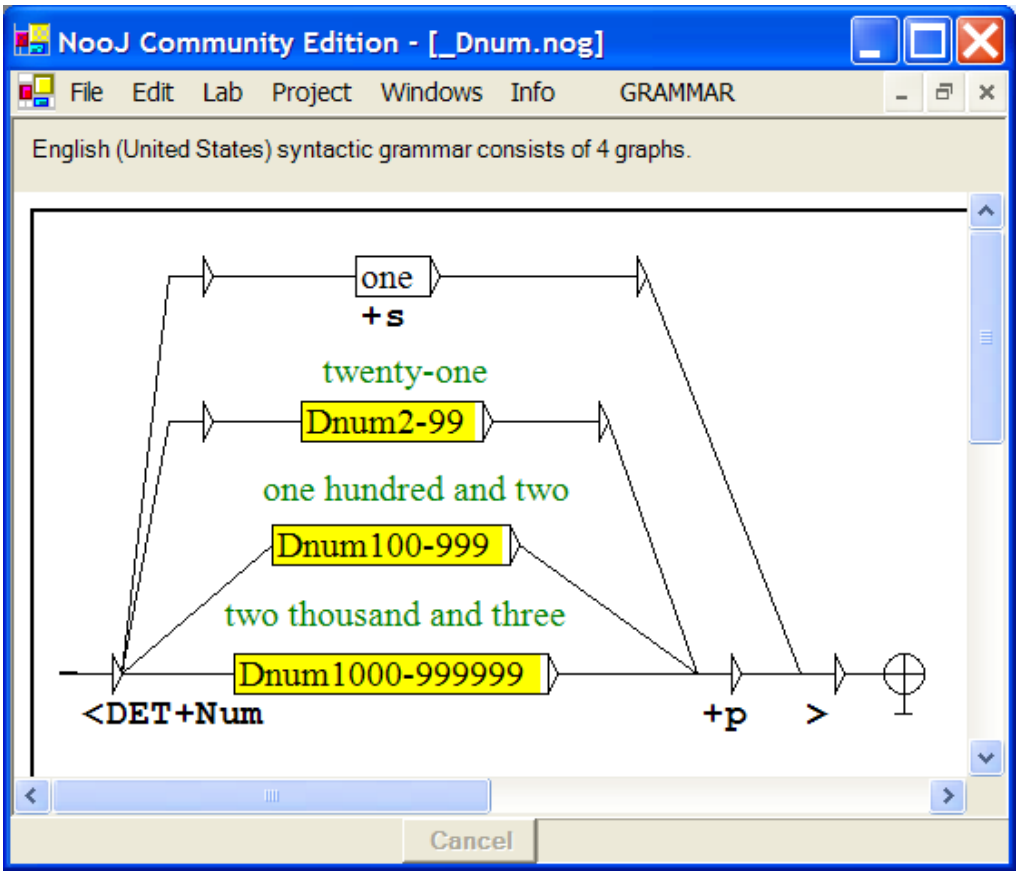


Figure 69. Syntactic grammar to annotate numeric determiners

Now, if this grammar is applied to the text (either manually via the **Locate** window, or automatically via **Text > Linguistic Analysis**), NooJ will process numeric determiners exactly as if they had been listed explicitly in a dictionary:

```
one,DET+Num+s
two,DET+Num+p
...
one hundred and twenty-two,DET+Num+p
...
seven thousand three hundred and two,DET+Num+p
```

and all numeric determiners in the text will be annotated accordingly:

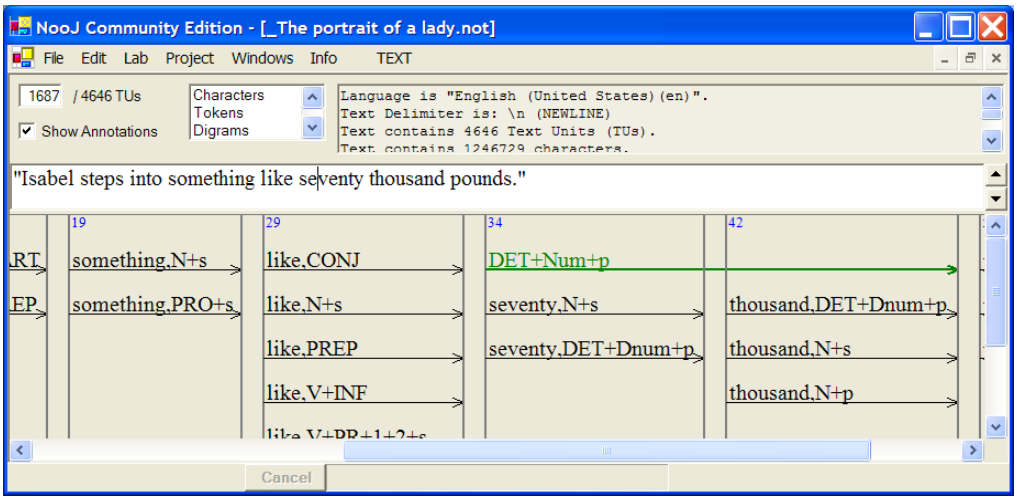


Figure 70. Annotating numeric determiners

Note that NooJ displays lexical annotations in black and syntactic annotations in green. You can apply syntactic grammars, and insert all their annotations into the TAS, either manually, or automatically:

(1) manually: apply the syntactic grammar via the **TEXT > Locate** command. Edit the resulting concordance, i.e. filter out sequences that were incorrectly recognized (**CONCORDANCE > Filter out selected lines**), then insert the newly created annotations in the TAS (**CONCORDANCE > Annotate Text**).

(2) automatically: select the syntactic grammar in Info > Preferences > Syntactic Analysis, then use the "+" and "-" buttons to assign it a priority over other ones. From now on, every time the command **TEXT > Linguistic Analysis** is being launched, NooJ's engine will apply it, as well as other syntactic grammars, and after all selected lexical resources.

Dates

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

A typical NooJ local grammar consists of a dozen graphs, and can be seen as a library of graphs; this allows different users to cooperatively build and share re-usable libraries of graphs.

For instance, consider the following graph **Tuesday** that recognizes names of days (*Monday ... Sunday*):

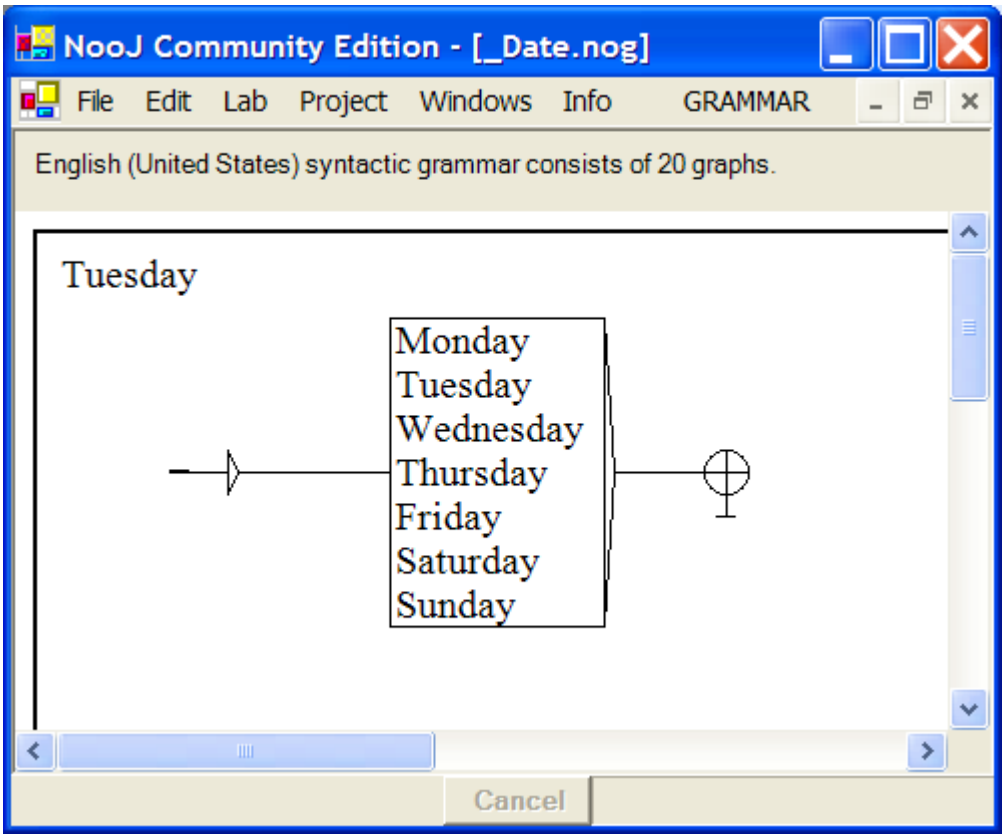


Figure 71. name of the days

This graph can be used in the grammar that recognizes more complex dates, e.g.:

last Monday; on Tuesday, June 5th; on Wednesday 31st

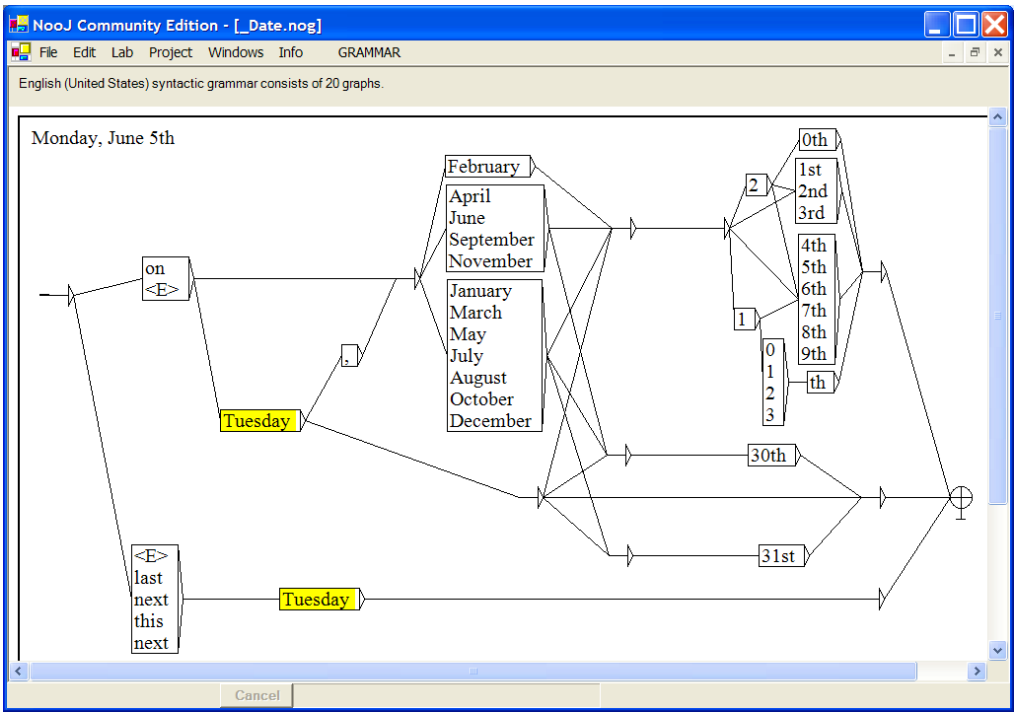


Figure 72. Graph Monday, June 5th

The graph **Monday, June 5th** can be reused, together with other graphs that recognize other expressions of date, as for instance the following graph **at 7:10 am**:

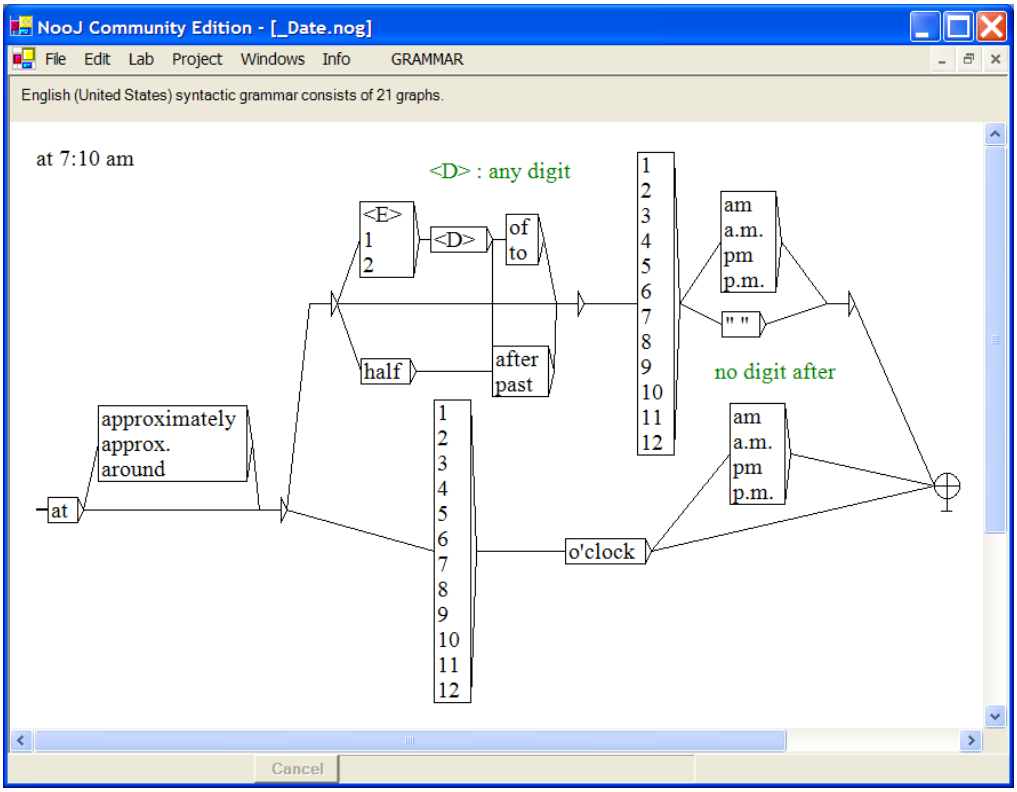


Figure 73. Graph at 7:10 am

These graphs, together with another dozen are then composed into a general grammar that recognizes and annotates complex date expressions:

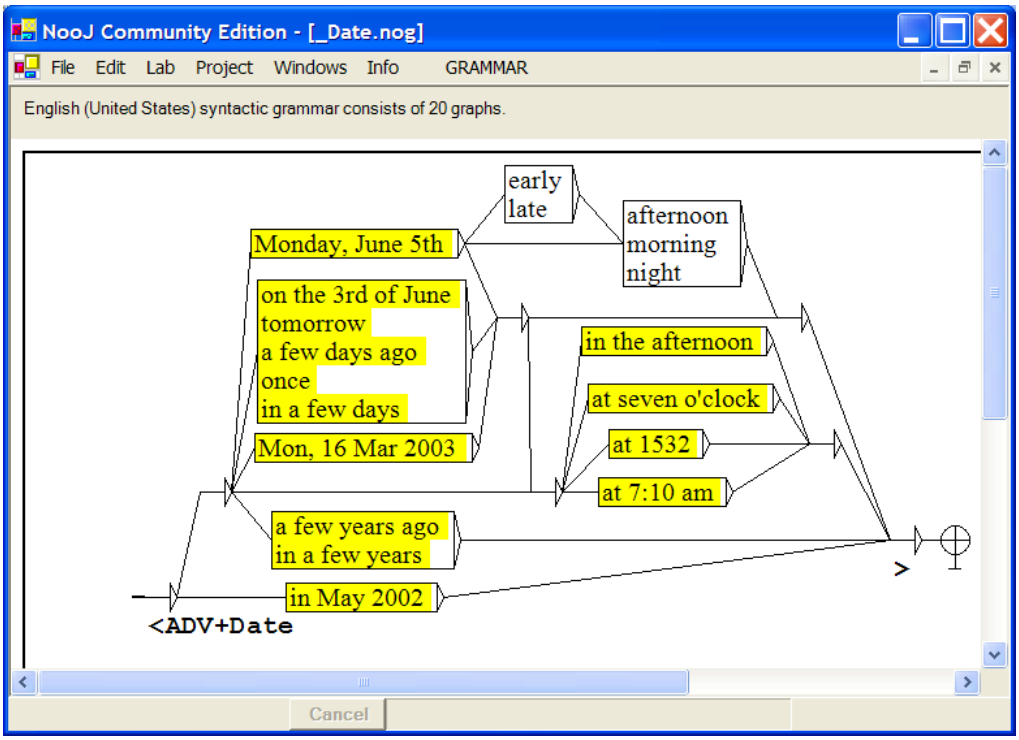


Figure 74. Grammar *Date*

If this grammar is applied to a text either manually, via **TEXT > Locate** and then **CONCORDANCE > Annotate Text**, or automatically, by selecting it in Info > Preferences > Syntactic Analysis, all dates in the text will be annotated as if they had been recognized by a dictionary lookup:

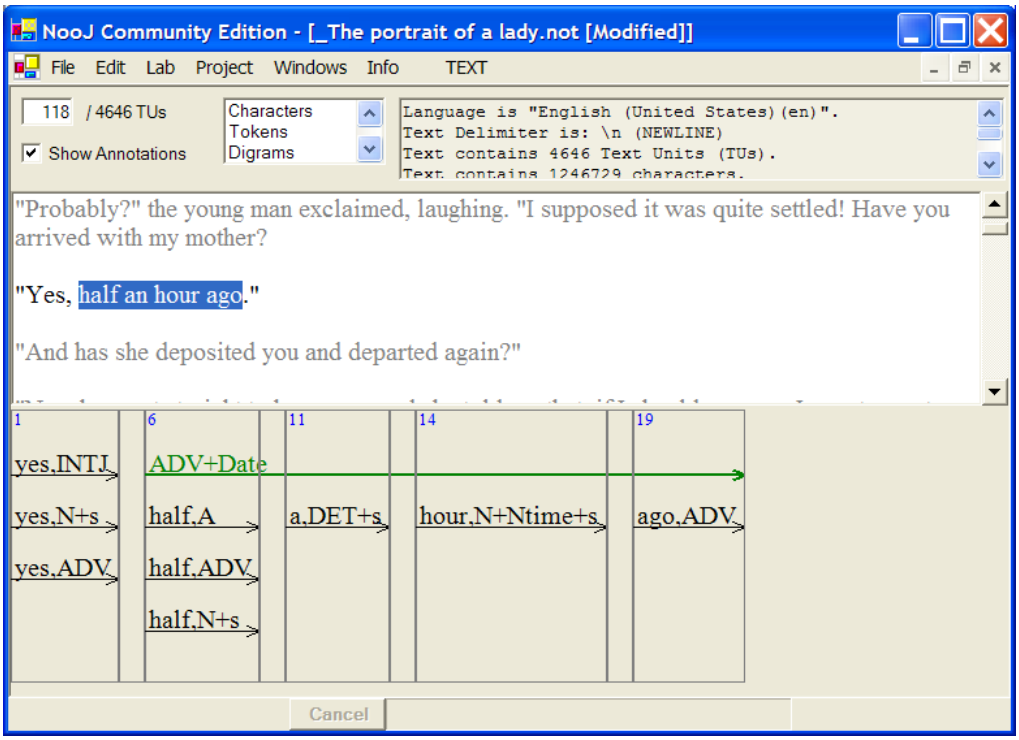


Figure 75. Annotating complements of date

Frozen and semi-frozen expressions are naturally described by local grammars, and should be applied automatically, by default, every time a text or a corpus is being analyzed with the command **TEXT > Linguistic Analysis**. In order to do that, select the grammars in **Info > Preferences > Syntactic Analysis**.

As soon as date expressions have been annotated, it is possible to access these annotations in NooJ regular expression queries, such as the following:

```
<PRO> <V+PP> <ADV+Date>
```

This would recognize phrases such as "*it happened on Monday, June 5th*", "*he left a few years ago*", etc.

NooJ syntactic grammars have access to all the text's lexical and syntactic annotations; they can add their own annotations, that will in turn be re-used in other grammars to describe more and more complex phrases, up to the sentence level.

Taking Contexts Into Account

Being able to specify exactly where annotations start and end allows also linguists to describe parts of the left and right contexts of the patterns they are looking for.

For instance, consider the following grammar:

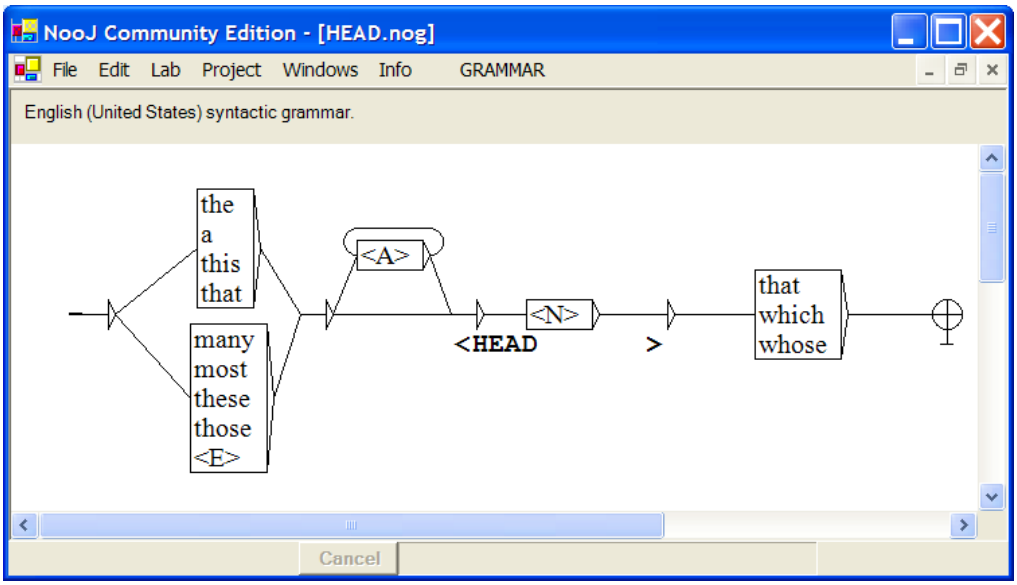


Figure 76. Annotating Noun Phrases' head

This grammar recognizes sequences such as "*Large red tables which*" and "*The American Import Export Organization that*", and then annotate the head of these incomplete noun phrases, respectively "tables" and "Organization".

Disambiguation = Filtering Out Annotations

We have seen how to *add* annotations to a text annotation structure. We also need methods to *remove* annotations from a Text Annotation Structure. NooJ provides three methods: automatic, semi-automatic and manual;

Automatic disambiguation

For instance, consider the following ambiguous text:

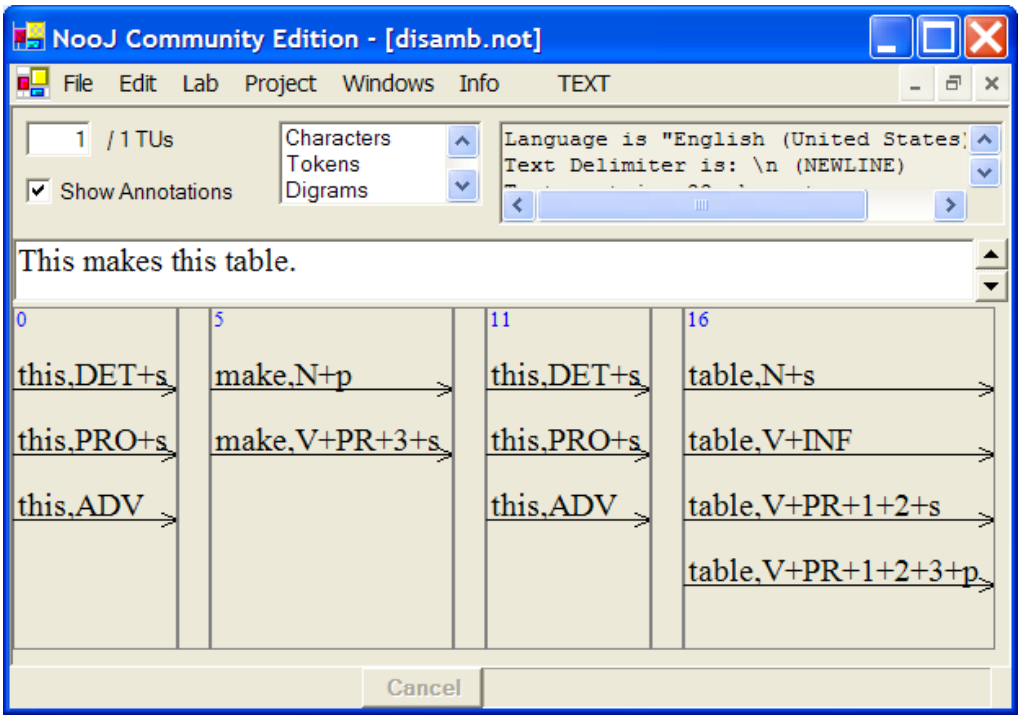


Figure 77. An ambiguous text

In fact, in this text, the first occurrence of the word "this" should correspond to a pronoun, and the second occurrence to a determiner; "makes" should be here annotated as a verb, and "table" as a noun.

Now consider the following grammar:

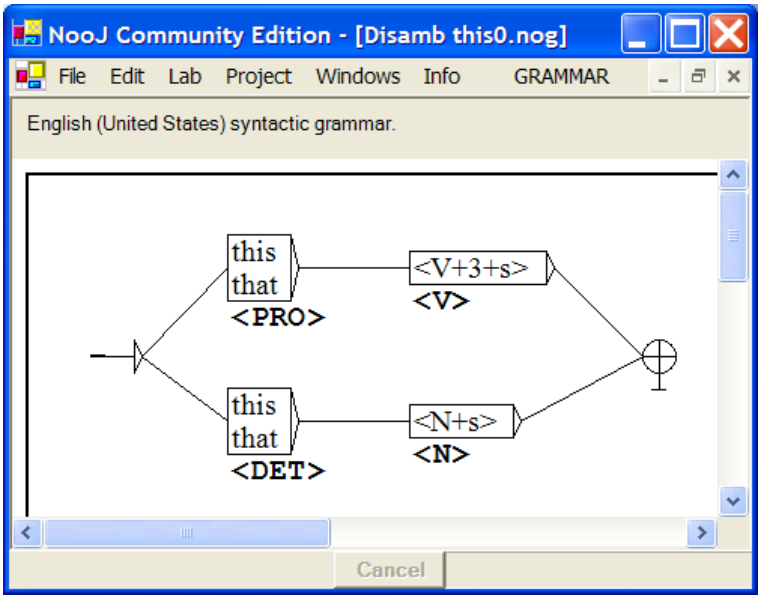


Figure 78. A Disambiguation Grammar

This grammar recognizes the sequence "*this makes*". In that case, it produces the annotation "**<PRO>**" at the position of the word "*this*", and the annotation "**<V>**"

at the position of the word "*makes*". Note that each of these annotations has a 0-character length; in consequence, NooJ uses them to filter out all the annotations of the Text Annotation Structure that are *not* compatible with them. In consequence, the two annotations "this,DET" and "this,ADV" are deleted, as well as the annotation "make,N+p".

In the same way, the grammar recognizes "*this table*", and produces the two filters "**<DET>**" and "**<N>**". In consequence, the annotations "this,PRO" and "this,ADV" are deleted, as well as the three annotations that correspond to the verb "to table". Applying this grammar to this text produces the following result:

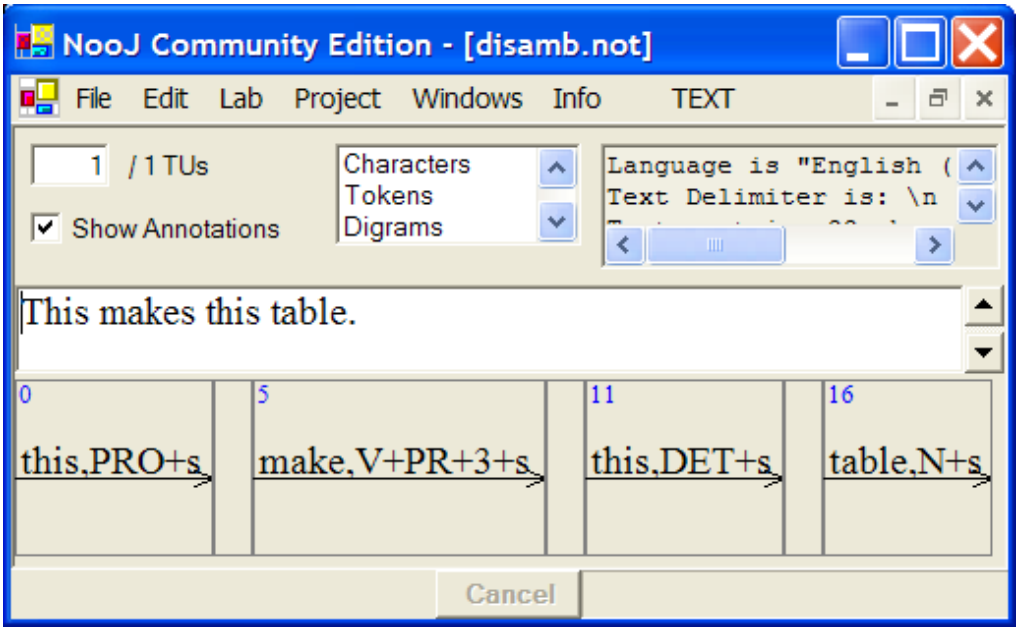


Figure 79. Disambiguated Text

Although this local grammar is very specific, it covers 2.5% of the text. Our goal is to develop a set of 30+ local grammars such as this one: target frequent ambiguous words to be as efficient as possible, while at the same time as specific as possible, so as to avoid producing incorrect results.

Semi-automatic disambiguation

The double constraint of efficiency and correctness is often very hard to follow, and in a number of cases, it is much too tempting to design very efficient rules which are correct in all but a few cases … For instance, it seems that all occurrences of the word *that* followed by a proper name correspond to the pronoun:

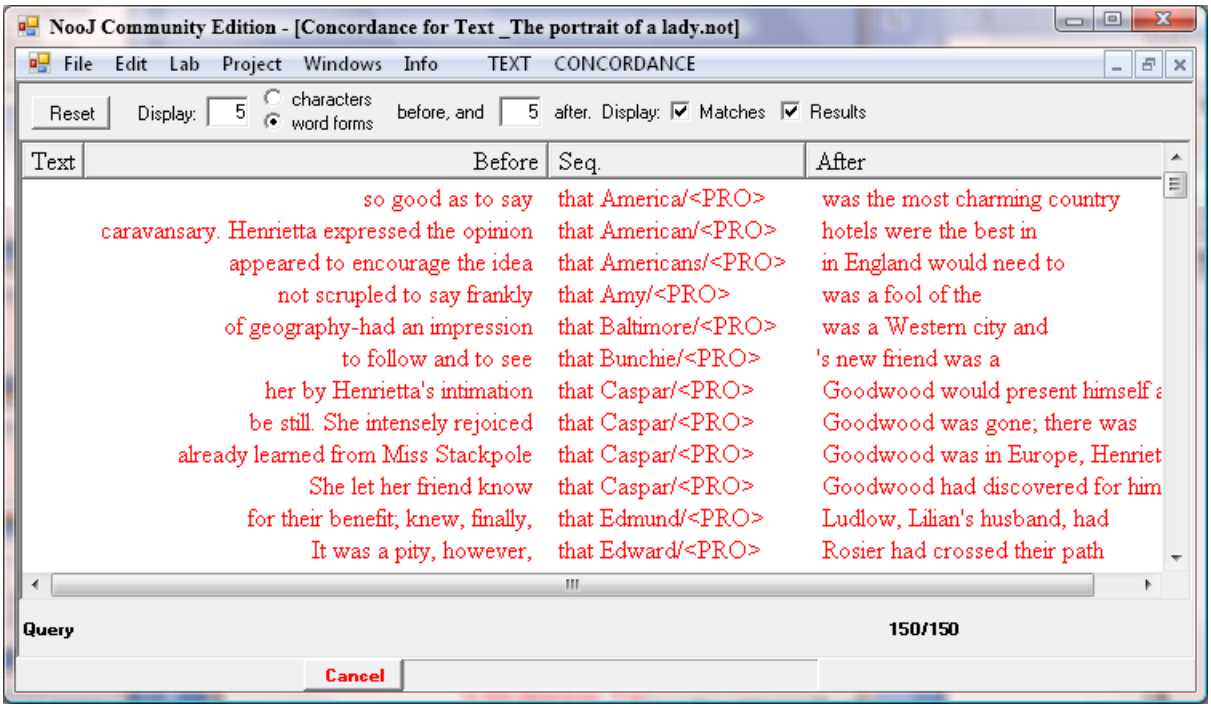


Figure 80. The context *that* <*N+PR*> is a good candidate for a disambiguation rule

This rule would have been very helpful to get rid of the incorrect concordance entries for the above-mentioned query <DET> <N>. Indeed, in the text *The portrait of a lady*, this rule can effectively be used to remove 150 ambiguities. However, it is not very difficult to build an example in which *that* is followed by a proper name, but is still a determiner:

Are you speaking of that John Doe?

Therefore, we need to be able to quickly enter simple disambiguation rules and check them before effectively applying them. In NooJ's new v2.2 version, it is possible to type in a query in the form of an expression associated with an output:

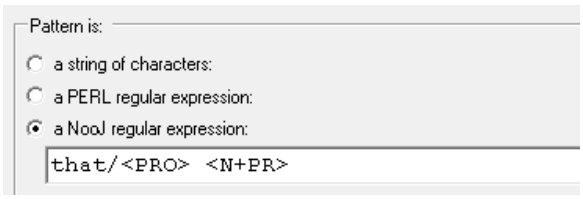


Figure 81. A simple disambiguation query

The regular expression **`that/<PRO> <N+PR>`** matches all the sequences of *that* followed by a proper name; then the output **`<PRO>`** is used to disambiguate the wordform *that*.

We can edit the resulting concordance in order to filter out incorrect matches, i.e. sequences that were recognized but should not be used. We first select the unwanted

concordance entries, then use the command **Filter out selected lines** of the CONCORDANCE menu to remove them from the concordance, and then Annotate Text (add/remove annotations) to perform the disambiguation for all remaining concordance entries.

This process is very quick: users should not hesitate to enter disambiguation commands that are very efficient but have poor accuracy, such as:

```
are/<V>
for/<PREP>
its/<DET>
```
etc.

The possibility of examining the resulting concordances and then filtering out incorrect analyses allows us to experiment more freely. Finally, note that we can save concordances at any moment: this allows us to validate large concordances at our own pace.

Editing the TAS

Sometimes, it is much faster to just delete unwanted annotations without having to write local grammars or even queries. In NooJ's new 2.2 version, we can just open the TAS (click **Show Text Annotation Structure** at the top of the text window), click an unwanted annotation, and then delete it either with the menu command **Edit > Cut**, or by pressing any of the following keys: Ctrl-x, Delete or Backspace. NooJ manages an unlimited number of undos: **Edit > Undo** or press Ctrl-Z.

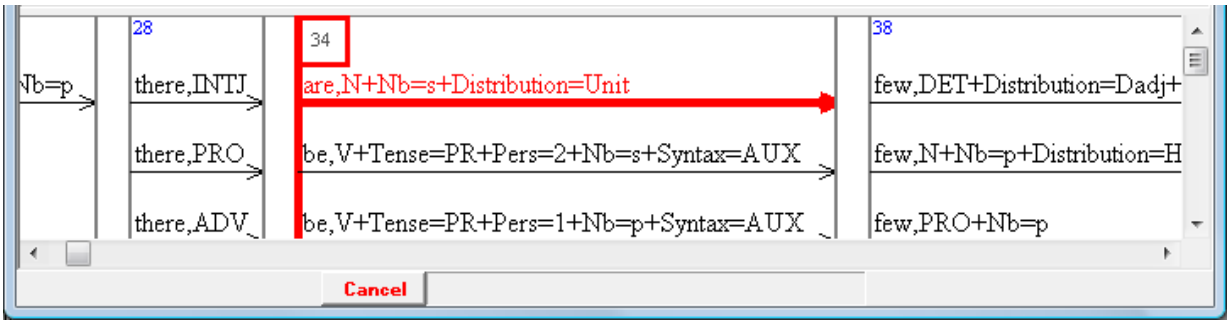


Figure 82. Editing the TAS

Examining the text's ambiguities

A new tool has been added to NooJ: the possibility of listing ambiguities according to their frequencies, to select one ambiguity and then to apply one solution to the text automatically. The most frequent ambiguous words will be the ones we intend to study in order to disambiguate the text efficiently.

Double-click **Ambiguities** in the text's result window, then click the header "**Freq**" to sort all ambiguities according to their frequency.

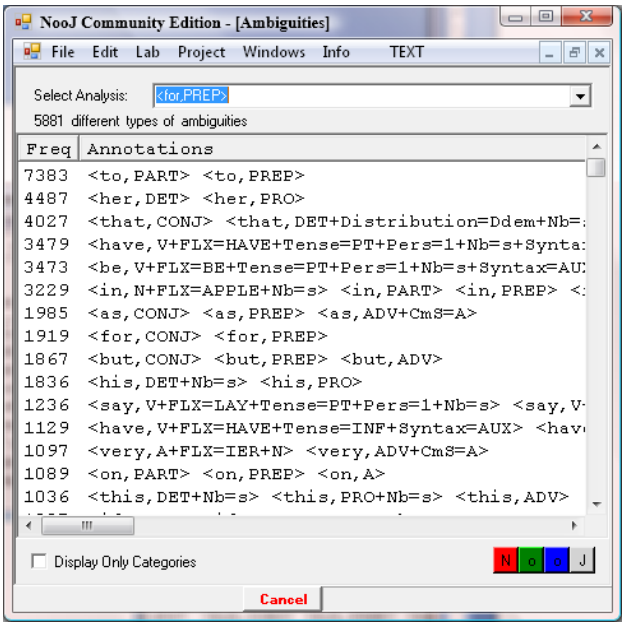


Figure 83. List of frequent Ambiguities

Here we notice that the wordform *for* is ambiguous and occurs 1,919 times in the text. We select the second disambiguation solution (at the top of the window): <for,PREP>. Then, we click one of the colored buttons to build the corresponding concordance. Finally, we filter out the rare cases in which *for* is actually a conjunction (I did not find any occurrence in the text *Portrait of a lady*) and then click Annotate text (add/remove annotations) to disambiguate the wordform.

Conversely, we can double-click Unambiguous Words in the text's result window in order to display the list of all unambiguous words. Usually, the wordforms *the, a* and *of* appear most frequently. We can then use frequent unambiguous words as anchors, to remove word ambiguities that occur immediately before or after them. For instance, the word *table* is theoretically ambiguous (it could be a verb), but in the sequences *the table* and *a table* it has to be a noun. Focusing on frequent unambiguous words can help us design new types of disambiguation rules.

Multiple Annotations

Up to now, sequences recognized by a local grammar (e.g. "twenty-three thousand seven hundred forty-one") were annotated as a whole (e.g. **DET**). We now show how to produce more complex grammars, to annotate only parts of the recognized sequences, or to produce more than one annotation for a given sequence.

Consider the following **Duration** grammar:

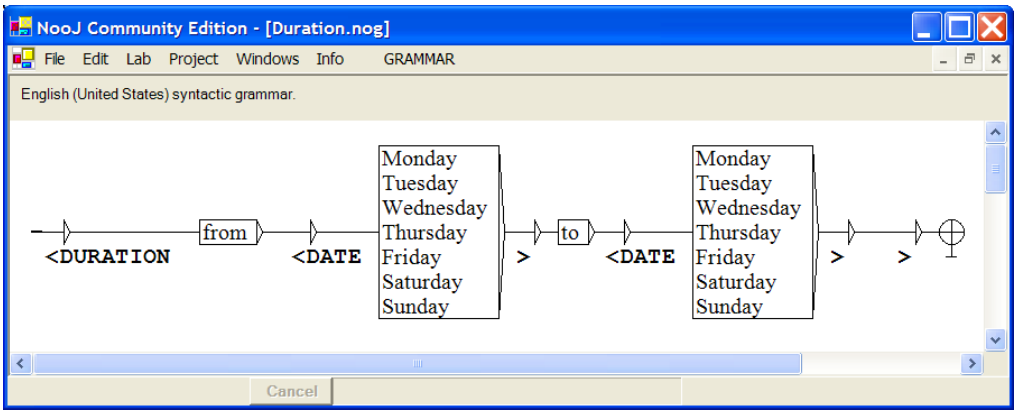


Figure 84. Grammar Duration

This grammar recognizes sequences such as "*From Monday to Thursday*". When recognized, sequences get annotated the following way:

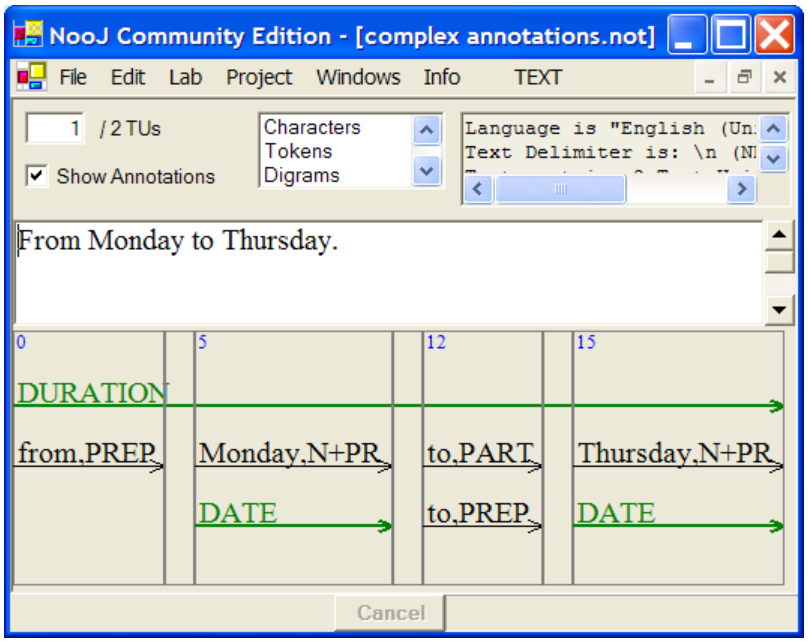


Figure 85. Complex annotations are inserted in the TAS

(Green annotations come from the above grammar; black annotations come from NooJ's lexical parser).

There is no limit to the number of embedded annotations that can be inserted at the same time for a given sequence. Indeed, annotations can (and should) be inserted locally, from inside embedded graphs. For instance, the following grammar:

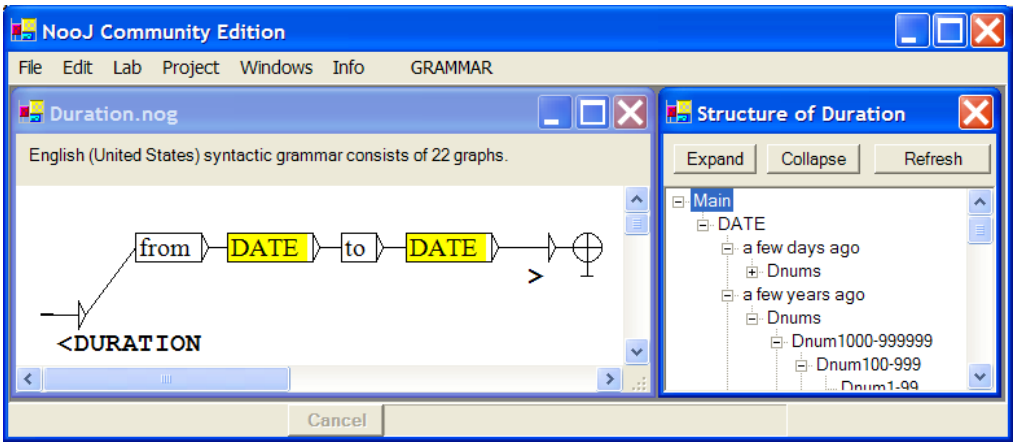


Figure 86. Embedded Grammars and Annotations

produces **DURATION** annotations; at the same time, the graphs **DATE** produce **DATE** annotations; they refer to other embedded graphs **TIME** which produce **TIME** annotations, etc.

A given grammar can add annotations, and remove others, at the same time. For instance, the following grammar can be used to disambiguate "this" and "that", and annotate the noun phrase at the same time.

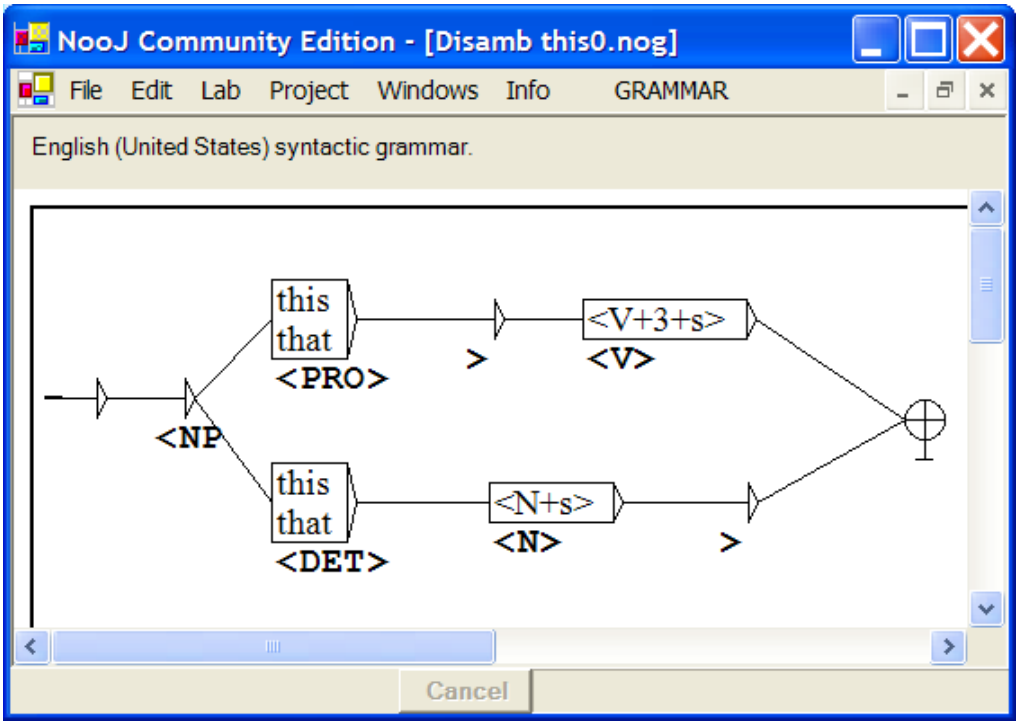


Figure 87. Adding and Removing Annotations at the same time

14. Frozen expressions

Affixes, simple words and multi-word Units are processed by NooJ's lexical parser, as we saw in the previous part. Frozen expressions, on the other hand, are discontinuous multi word units that cannot be recognized by NooJ's lexical parser. NooJ provides two methods to formalize frozen expressions:

-- NooJ's local grammars can represent and recognize all the possible contexts of a given expression.

-- NooJ allows linguists to link a dictionary that describes the possible components of the frozen expression, with a grammar that describes its syntactic behavior[6].

Basically, linguists typically use the first, easiest, method to represent limited series of frozen expressions or semi-frozen expressions (such as named entities), whereas they use the second, more powerful tool to represent large numbers of expressions, typically lists of hundreds of idioms, proverbs, etc. that are best represented by dictionaries.

Local grammars of frozen expressions

NooJ's syntactic local grammars are used to represent, automatically recognize and annotate expressions, such as the ones we saw in **0Local grammars**. Complex determiners, such as the ones represented by the grammar in Figure 69. 142) can indeed be considered as full **ALU**s.

[6] Linking a dictionary with a grammar is natural in linguistics: Maurice Gross has indeed described frozen expressions in "lexicon-grammar tables", which are dictionaries in which lexical entries are associated with some syntactic properties. Lexicon-grammar tables can easily be imported into NooJ, and any NooJ dictionary can be displayed as a lexicon-grammar table: the commands "Display As Table" and "Display As List" perform the conversion. Here, we are taking a further step: NooJ uses the grammar to validate the dictionary's entries.

The situation is a little more complex when we deal with discontinuous **ALU**s. For instance, the phrasal verb *to give up* occurs in the two following sentences:

John **gave up** *his rights*. *John* **gave** *his rights* **up**.

Therefore we need to represent both ALUs with an annotation. Consider the following grammar:

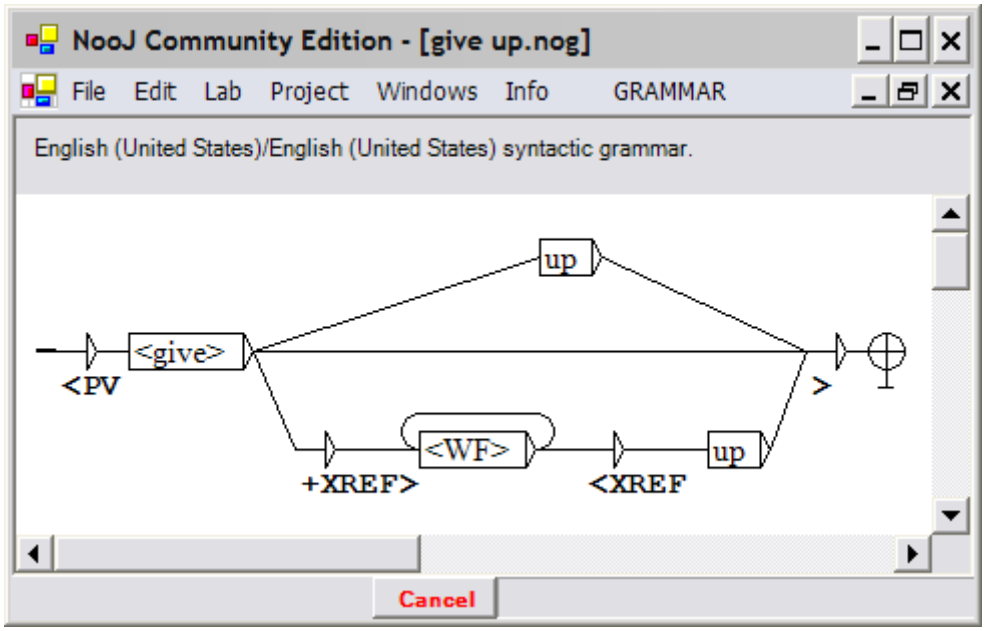


Figure 88. A discontinuous expression

This grammar recognizes the sequence "gave up", and then produces the single annotation <**PV**> (Phrasal Verb). The grammar recognizes also the sequence "gave his right up", and then produces two annotations that contain the keyword XREF. This keyword tells NooJ to aggregate these annotations. The result is a discontinuous annotation, as seen in the following figure:

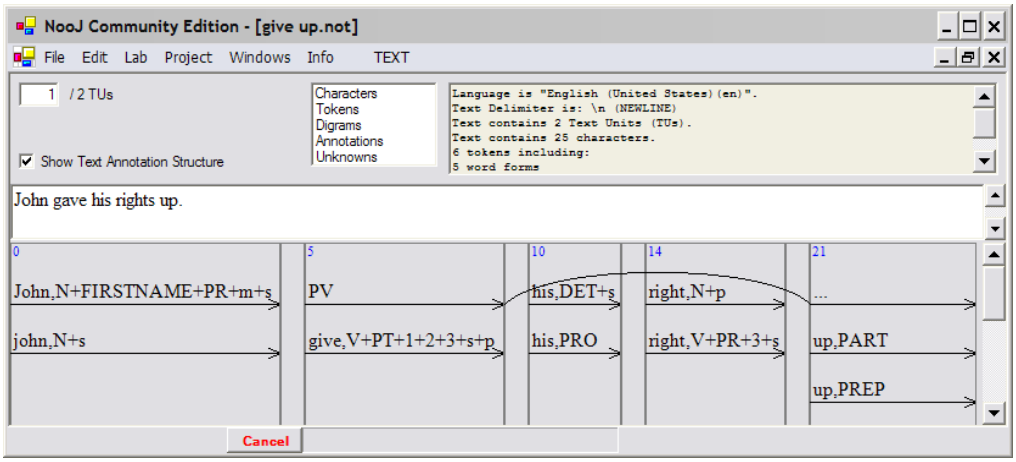


Figure 89. A Discontinuous Annotation

In other words, the expression "gave ... up" has been annotated as a <**PV**>. The previous local grammar can be enhanced as one might decide to annotate the expression in a richer way: instead of annotating the whole expression as a <**PV**>, one might rather annotate the first component of the expression as <**V+PV**> and the second component as <**PART+PV**>:

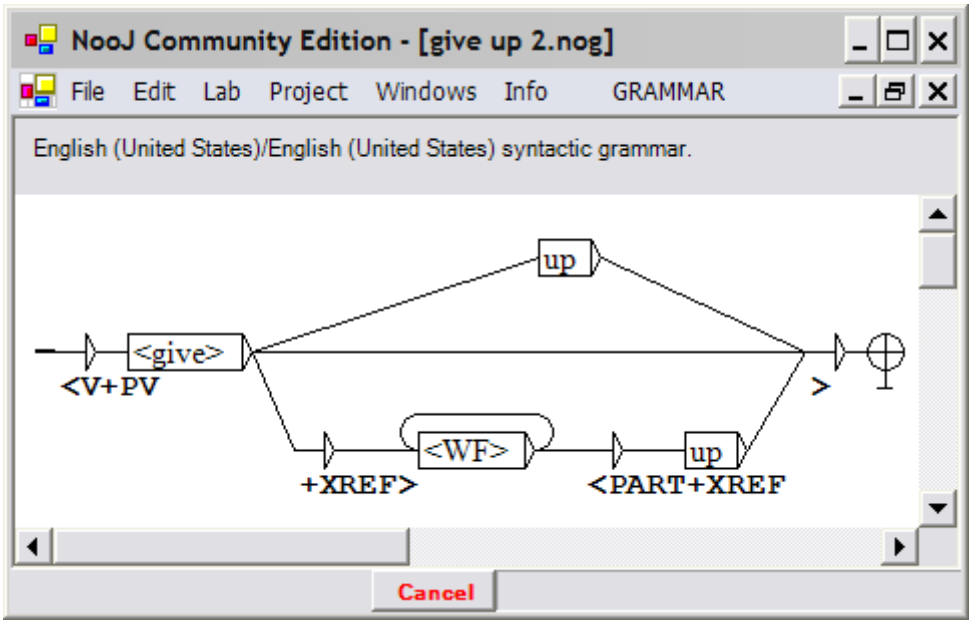


Figure 90. A Richer Grammar

In that case, the resulting text annotation structure will be the following:

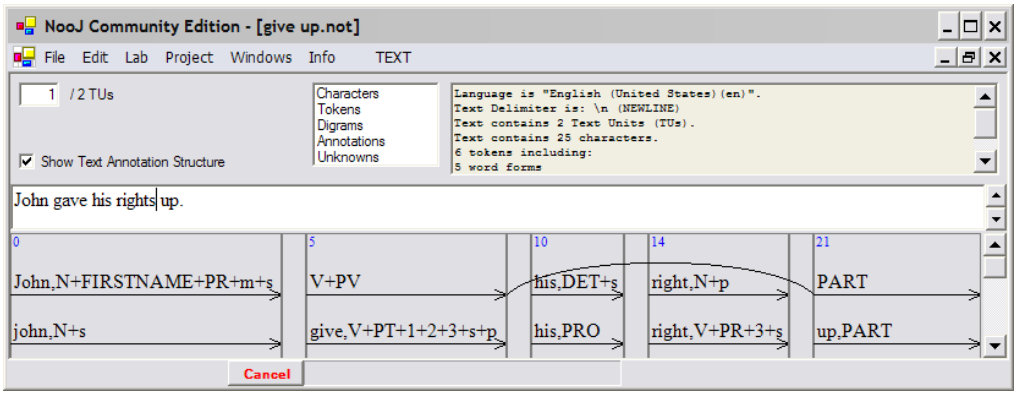


Figure 91. A Richer Annotation

The richer annotation allows NooJ to still be able to find "gave" as a verb, and "up" as a particle, even though the whole can also be processed as a single entity.

One might even decide to lemmatize the annotation, and to describe it by reusing parts of the lexical information associated with "gave" so that the corresponding ALU will be **<give up,PV+tr+Pret>**. In order to do that, we need to add to the previous

grammar a variable, say **$V**, that will hold the lexical information that is produced by the node <give>.

In a "real" linguistic application, we would want to replace the node <WF>* (which matches any sequence of tokens) with a more adequate description, such as a description of noun phrases. There are two possibilities here:

-- We could insert a simplified description of NPs in an embedded graph. To do that, just replace the node <WF> with a :NP node. Note that the :NP graph itself can also have its own embedded graphs.

-- There might already be a general syntactic grammar that annotates all noun phrases as <NP>. In that case, make sure that the **NP** grammar has been selected in Info > Preferences > Syntactic Analysis, then simply replace the node <WF>* with a node <NP>.

Linking dictionaries and grammars

Local grammars are easy to implement when one wants to describe a small family of expressions, but they would be inadequate if one wants to describe several hundred expressions.

In that case, we need to link a dictionary to a syntactic grammar. This association between a dictionary and a grammar is very natural; indeed linguists at the LADL and in other labs have used lexicon-grammars to represent frozen expressions for a long time[7]. In order to formalize a series of frozen expressions in NooJ, we need to construct a dictionary and a grammar, and link them.

(1) The dictionary contains the **characteristic component** for each expression.

The characteristic component of a frozen expression is a simple word or a multi-word unit that occurs every time the frozen expression occurs.

In general, the characteristic component triggers the recognition for the whole expression. For instance, both the verb *take* and the noun "account" could be characteristic components of the expression *to take into account*. However, the verb *be* cannot be a characteristic component of the expression "to be late", because this expression occurs even if the verb *be* does not occur, for instance in:

John made Mary late.

[7] NooJ's dictionaries of frozen expressions can easily be constructed semi-automatically from lexicon-grammar tables.

(which is analyzed as *John caused Mary to be late*). In that case, the characteristic component should be *late*.

For this reason, the characteristic component of the associations of a predicative noun and its support verb (and all the support verb's variants) should always be the noun.

When choosing the characteristic component for a frozen expression, it is important to avoid words that occur in a lot of frozen expressions: if "take" is the characteristic component of thousands of expressions (e.g. *take into account*, *take care of*, *take place*, *take a swim*, etc.), there will be thousands of entries "take" in the dictionary; solving all these ambiguities will slow NooJ's lookup procedure considerably. It is much better to choose *really characteristic* components, such as *into account*, *care of*, *place*, *a swim*.

(2) In the dictionary, each characteristic component must be associated with each of the other components of the expression, as well as some properties that will be checked by the associated grammar. For instance, we could have the following lexical entry:

```
take,V+FLX=TAKE+PP="into account"
```

The **FLX** information ensures that the characteristic component will be recognized even when inflected. The **PP** information links the lexical entry to the word sequence "into account". More information, such as **N0Hum** (to ensure that the subject of the frozen expression is a Human noun phrase), could be added to enhance the precision of the description.

(3) The associated grammar must describe the context in which the frozen expression occurs. It includes the characteristic component, as well as the other components of the expression. Matching the grammar validates the frozen expression, and its output is usually used to annotate the expression.

For instance, the following grammar can be associated with the dictionary above to recognize occurrences of the frozen expression "to take ... into account".

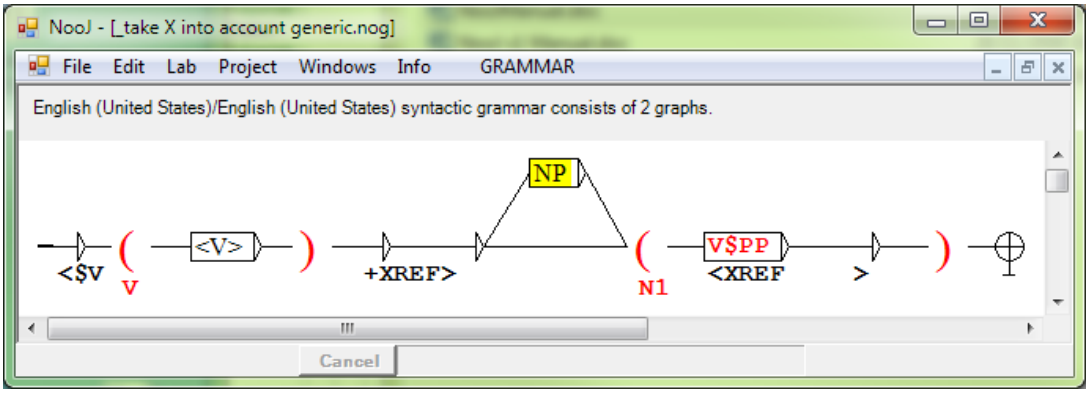


Figure 92. A generic syntactic grammar

The compound variable **VPP** (in red) links the verb recognized by <V> with the value of its **PP** field value. In other words, when the grammar is activated by *take* (which has the field +PP="into account"), the grammar behaves exactly as the following one:

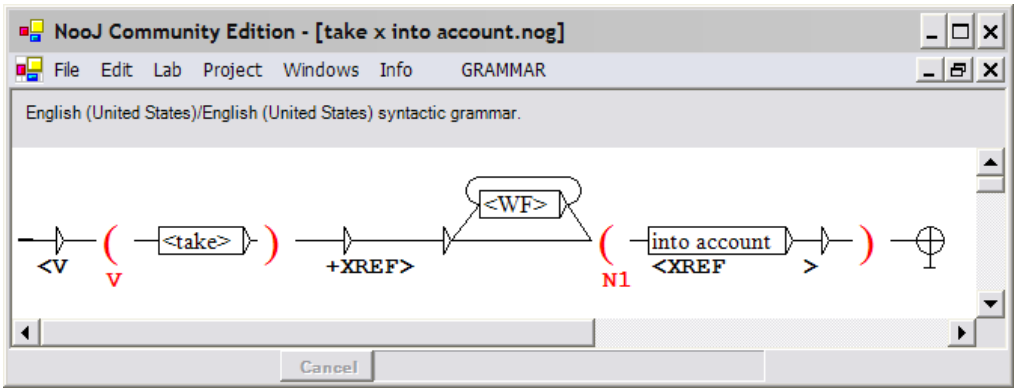


Figure 93. Instantiation of a generic syntactic grammar

The advantage of the generic grammar is that it needs to be written only once, even if the corresponding dictionary has thousands of lexical entries.

In order to link a dictionary and its corresponding grammar, just give them the same name, and store them in the Lexical Analysis folder. For instance, the French Lexical Analysis module contains a pair of files C1D.nod, C1D.nog that formalize over 1,600 frozen expressions like "perdre le nord" (= *to lose it*). This

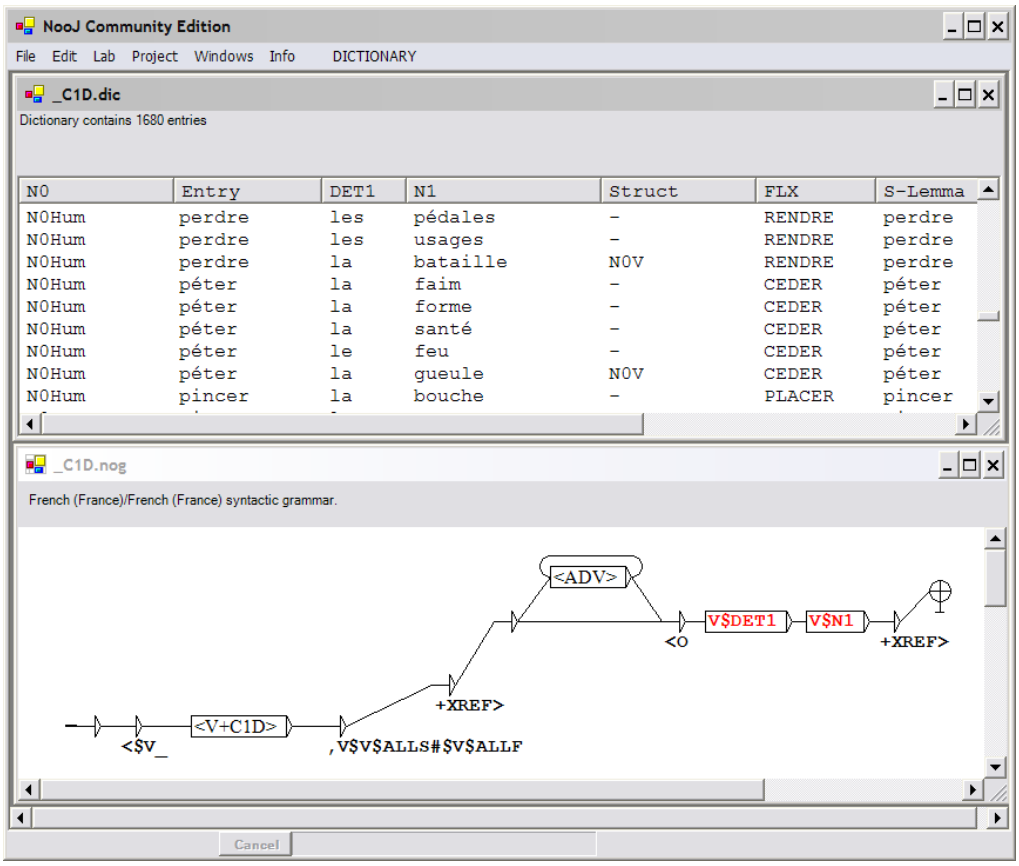


Figure 94. Frozen expressions formalized in a pair Dictionary / Grammar

NooJ's **C1D** pair of dictionary, grammar formalizes Maurice Gross's **C1D** lexicon-grammar table.

15. Rules of grammars' application

NooJ's syntactic parser follows a set of rules.

A grammar cannot recognize an empty node

NooJ's syntactic parser cannot apply grammars that recognize the empty string. For instance, consider the following grammar:

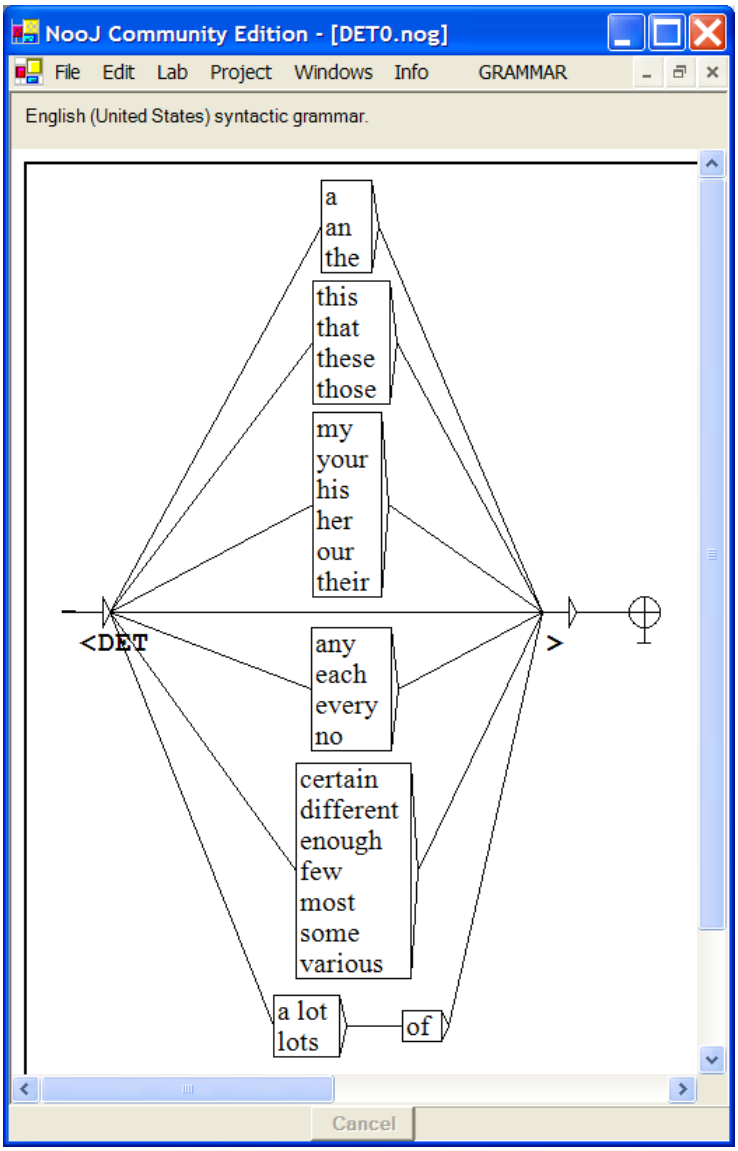


Figure 95. graph recognizing the empty string

This graph recognizes and annotates a few determiners, including the empty determiner that can be found in the following text:

Birds have wings

Unfortunately, one cannot directly apply this grammar to a text: doing so would request NooJ to insert an infinite number of **<DET>** annotations at every position in the text!

If you try to apply to a text a grammar that recognizes the **empty string**, NooJ will recognize the problem, will warn you and then abort.

In Syntax as well as in Morphology, there are a number of cases in which it is important to describe sequences of grammatical words that are implicit; for instance, the

sequence "who is" can arguably be seen as implicit in the noun phrase "John, late as usual".

NooJ provides ways to formalize empty strings, i.e. to associate them with linguistic information: even though a grammar cannot recognize the empty string, it can contain an embedded graph that recognizes the empty string. In the previous example, we could include the DET0 graph in a more general NP ("noun phrase") grammar, as long as the NP grammar itself does not recognize the empty string. We will also see how that implicit words or sequences can be formalized as "default values" for grammar variables (see below).

Ambiguity

When NooJ's parser produces more than one analysis for a particular sequence, the sequence is said to be ambiguous. For example, consider the following grammar, which annotates "this" or "that" as pronouns if they are followed by a verb, and as determiners if they are followed by a noun.

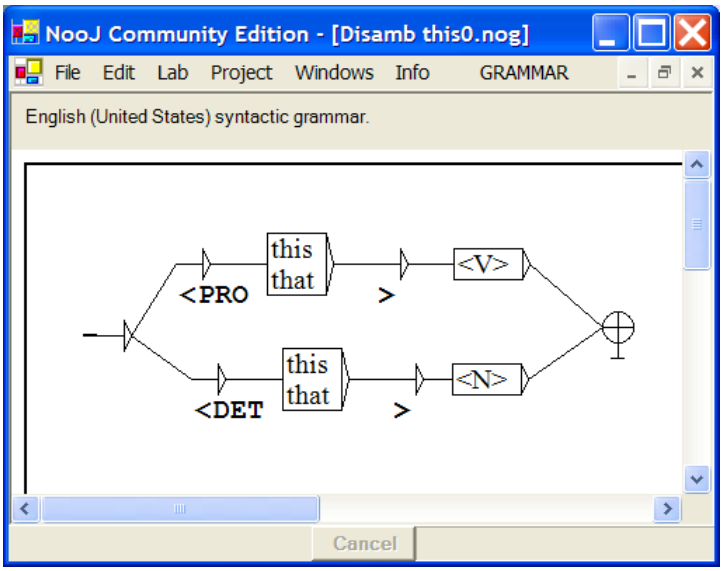


Figure 96. an ambiguous grammar

But what happens if the following word is ambiguous (noun or verb), such as "rose" in:

... this rose yesterday (rose = to rise,V) ...
... this rose is beautiful (rose = N) ...

In that case, NooJ produces both results, i.e. two lines in the concordance:

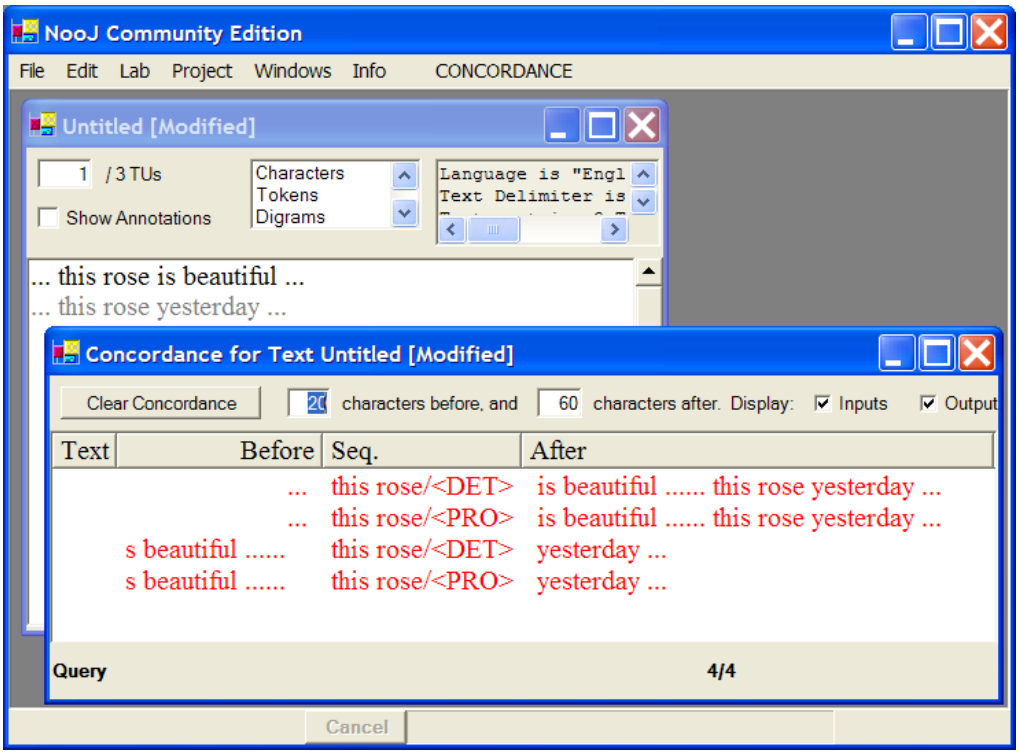


Figure 97. An ambiguous annotation

Now, if the user chooses to insert the annotations into the Text Annotation Structure (CONCORDANCE > Annotate Text), we get both annotations:

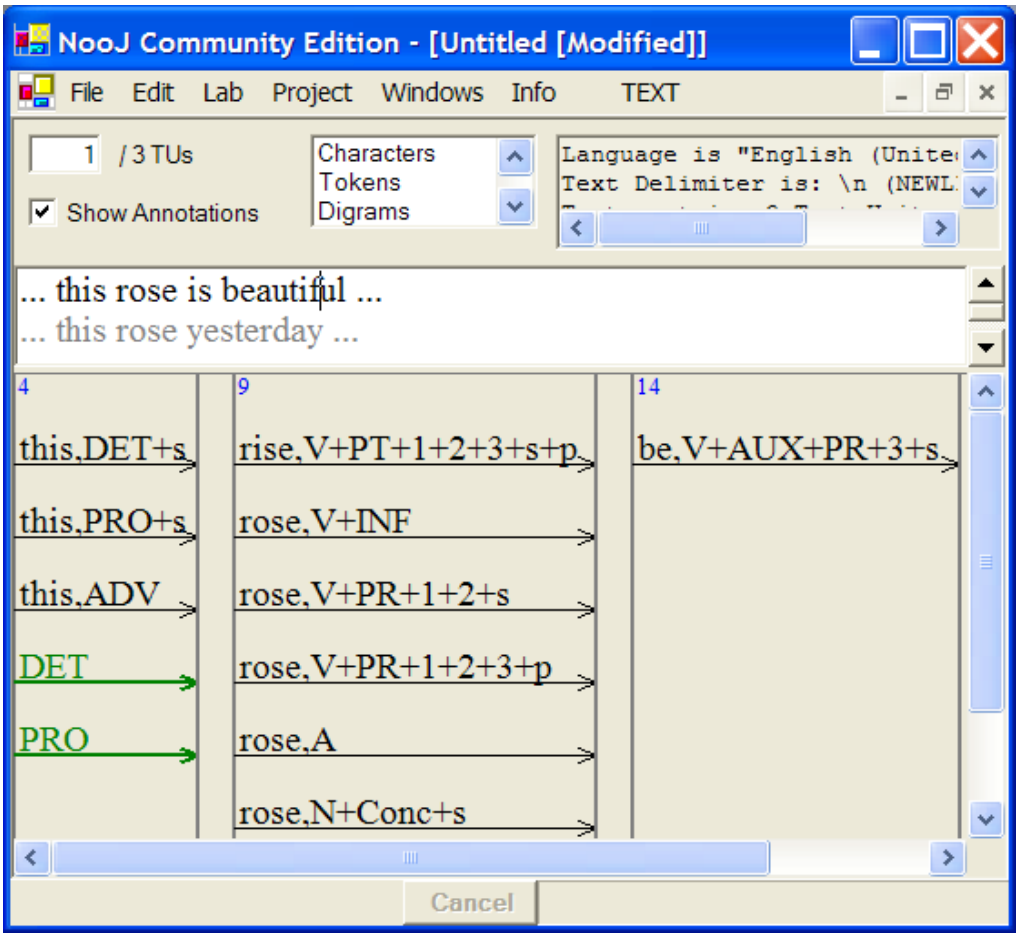


Figure 98. an ambiguous annotation

Caution: In general, NooJ cannot detect such contradictory commands. It is your responsibility to verify that your grammar, regardless of the degree of complexity, does not associate different annotations for one given sequence (except of course if the sequence is genuinely ambiguous!).

Ambiguous matches with different lengths

A given grammar could potentially recognize more than one sequence at one position, and then associate each sequence with a different annotation. For instance, consider the following query, entered as a regular expression:

```
<be> <WF>* <N>
```

If entered in the Locate panel, this expression will locate in texts all the sequences constituted of a conjugated form of the verb to be, followed by any number of wordforms (including 0), followed by a noun, e.g. "was admitted in a hospital".

The symbol <WF> recognizes any wordform, including nouns. So what happens if there are several nouns in a given text unit, such as in:

(T) *... there are few* **hours** *in* **life** *more agreeable than the* **hour** *dedicated to the* **ceremony** *known as* **afternoon tea**. *There are circumstances...*

Shortest / Longest / All matches

In the **Locate** panel, the user can choose one of the three following options:

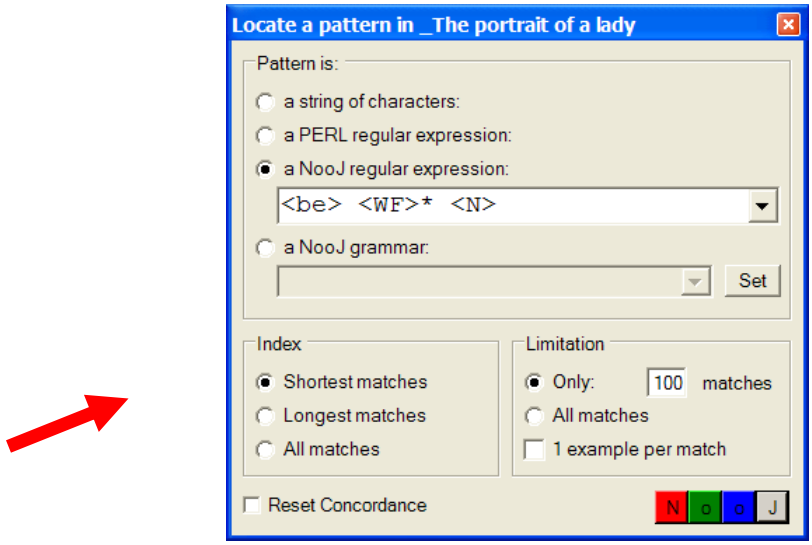


Figure 99. Grammars' application options

These options decide what sequences NooJ should take into account in case of ambiguities, i.e. what sequences will show up in the Concordance.

Index Shortest matches

This option tells NooJ to stop as soon as it gets a match. Therefore, the concordance for text (t) will show the shortest sequence:

... there **are few hours** *in life more agreeable...*

In that case, it is as if <WF> meant "any wordform but a noun".

Index Longest matches

This option tells NooJ to keep going as far as possible, and produce only the longest sequence possible. Therefore, the concordance for text (t) will show:

... there **are few hours in life more agreeable than the hour dedicated to the ceremony known as afternoon tea** *. There are circumstances...*

Note that it is important to keep text units small; if the text was opened as one single text unit, a query such as this one could produce a sequence of hundreds of pages!

Index All matches

This option tells NooJ to produce all possible matches. Therefore, the concordance for text (t) will display five entries:

... *there* **are few hours** *in life more agreeable than the hour dedicated to the ceremony known as afternoon tea. There are circumstances...*

... *there* **are few hours in life** *more agreeable than the hour dedicated to the ceremony known as afternoon tea. There are circumstances...*

... *there* **are few hours in life more agreeable than the hour** *dedicated to the ceremony known as afternoon tea. There are circumstances...*

... *there* **are few hours in life more agreeable than the hour dedicated to the ceremony** *known as afternoon tea. There are circumstances...*

... *there* **are few hours in life more agreeable than the hour dedicated to the ceremony known as afternoon tea** *. There are circumstances...*

Ambiguity and the insertion of annotations

Consider the following grammar:

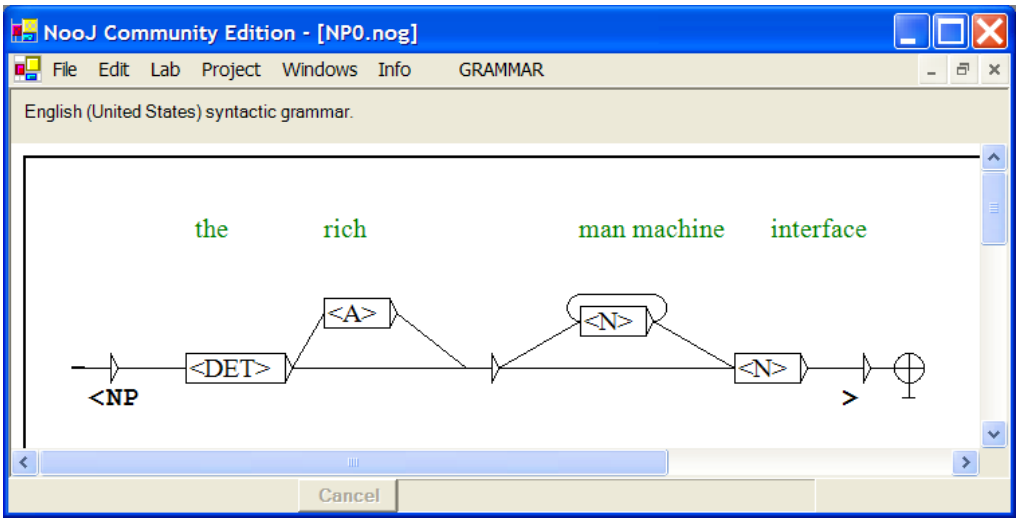


Figure 100. An ambiguous grammar

This grammar recognizes sequences such as "the table" by following the direct path <DET> <N>, "the red table" (through <DET> <A> <N>), "the American imports exports companies" (through <DET> <A> <N> <N> <N>), etc. Recognized sequences are annotated as **NP**.

Now, more specifically: what happens exactly when the sequence "The American imports exports companies" is parsed? Note that "American" can be an Adjective or a Noun, "imports" and "exports" can be nouns or verbs, and "companies" is a noun. Therefore, the grammar above recognizes actually four sequences:

```
(a) The American
(b) The American imports
(c) The American imports exports
(d) The American imports exports companies
```

As for regular expressions queries, the **Locate** panel provides the three options:

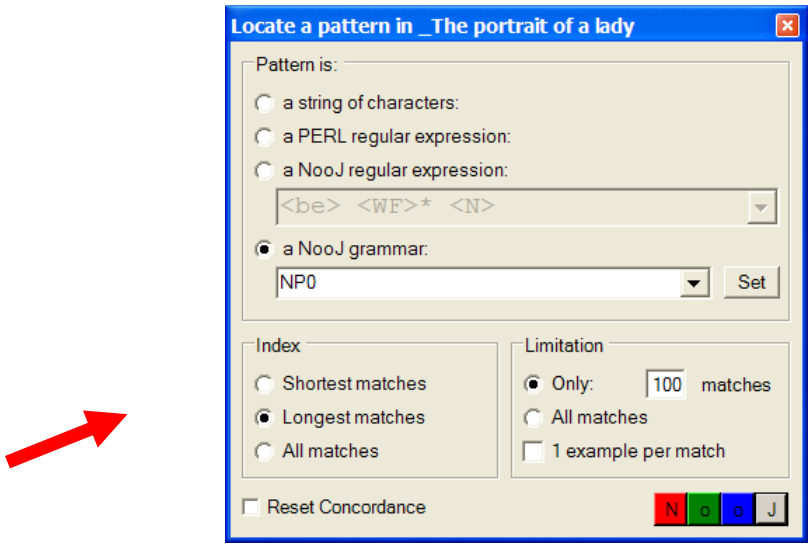


Figure 101. Grammars' application options

These options decide what sequences NooJ should take into account, and therefore what annotation will be inserted in the TAS.

Index Shortest matches

In that case, NooJ's parser stops as soon as it has found a match. In our example, when NooJ reads "The American", it considers it a match thanks to the path **<DET>** <**N**>, and then simply ignores all the other matches. In other words, the resulting concordance will only contain the first sequence (a): "The American". If the user decides to insert the concordance's annotations in the Text Annotation Structure (**CONCORDANCE > Annotate Text**), the resulting TAS will include only one **NP** annotation:

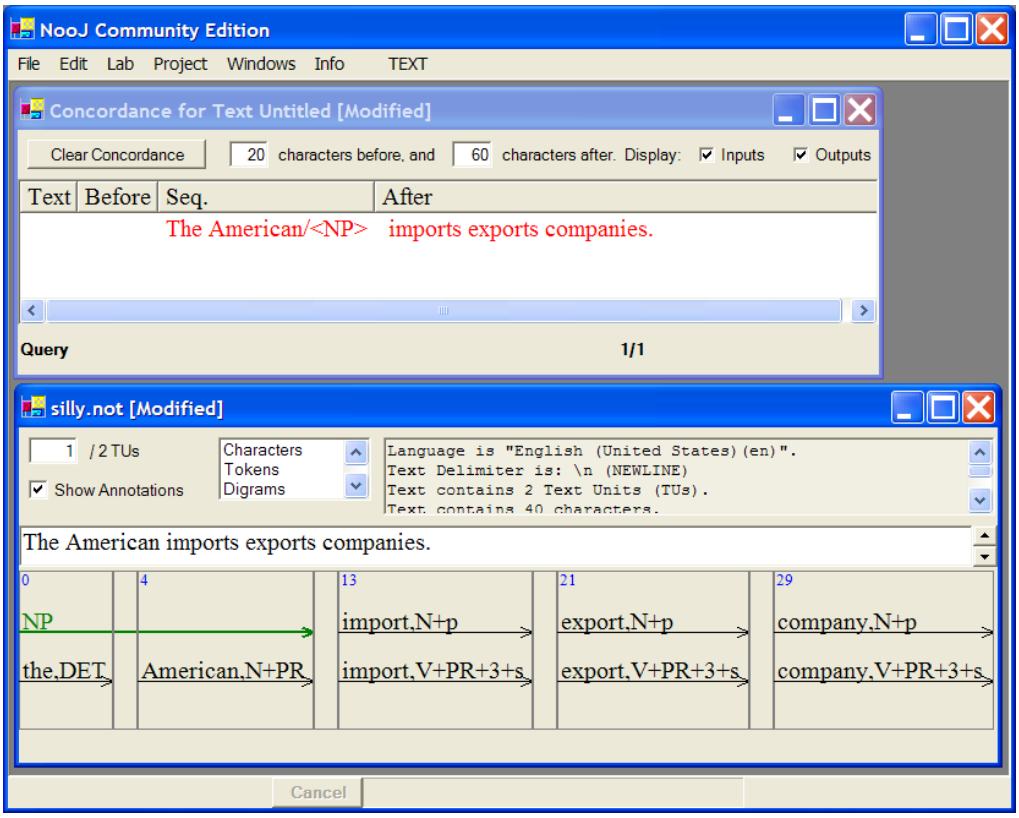


Figure 102. the shortest match

Note that even if the option "Index Shortest Matches" is selected, it is still possible that there are more than one "shortest sequence", associated with different annotations. In that case, NooJ will annotate all solutions.

Index Longest matches

In that case, NooJ's parser returns only the longest matches, and simply forgets the shorter ones. In our example, NooJ recognizes "The American import export company" thanks to the path **<DET> <A> <N> <N>** <N>, and ignores all the shorter matches. The resulting concordance will only contain the last sequence (d): "The American import export company"; the resulting annotated text will contain only the longest annotation:

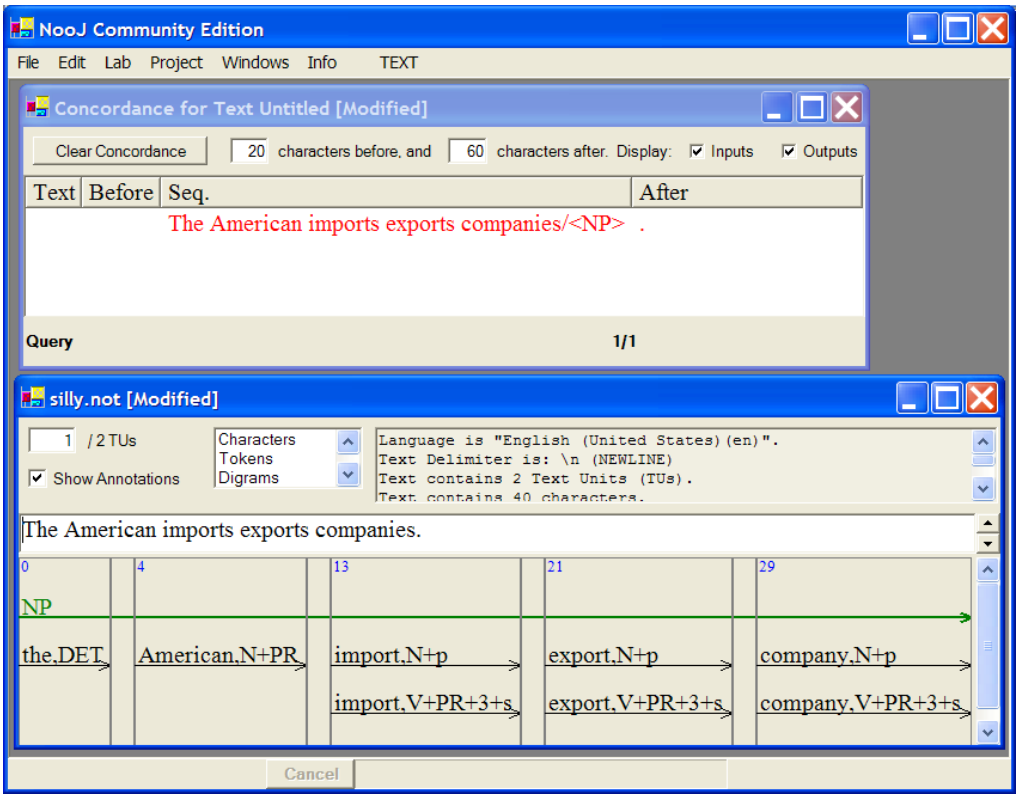


Figure 103. the longest match

Note that even if the option "Index Longest Matches" is selected, it is still possible that there are more than one "longest sequence", associated with different annotations. In that case, NooJ will annotate all solutions.

Index All matches

In that case, NooJ's parser returns all matches, in our example: four results. The resulting concordance will contain the four lines (a), (b), (c) and (d). If we ask NooJ to insert the new annotations in the text (**CONCORDANCE > Annotate Text**), we get the following TAS:

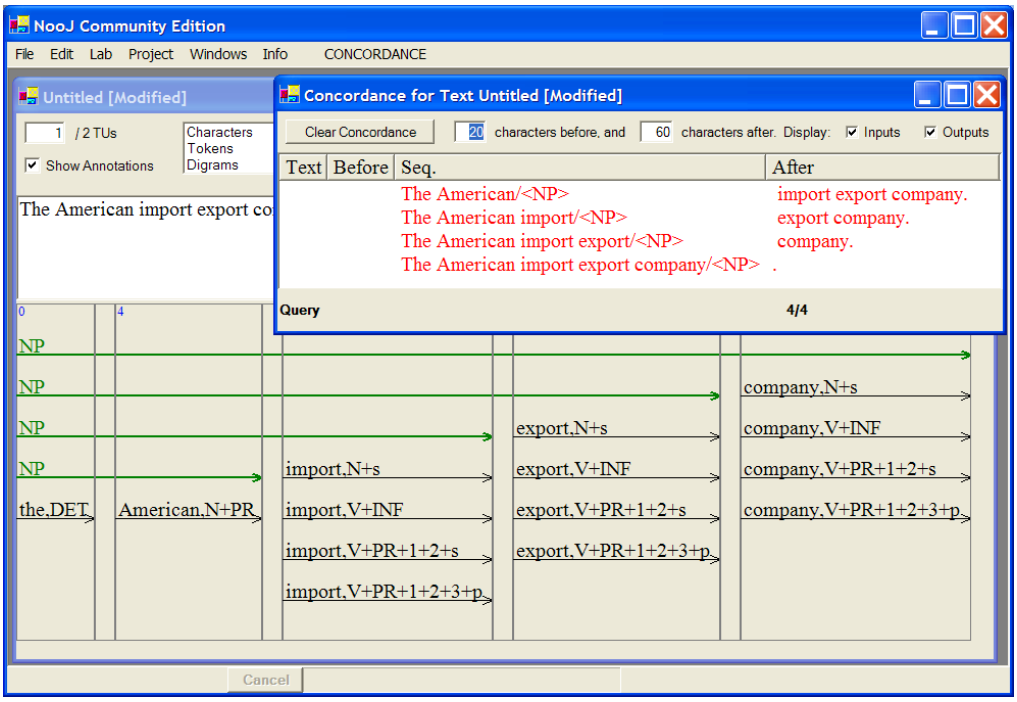


Figure 104. an ambiguous NP

Overlapping matches

More complex cases of ambiguity may occur. Consider for instance the following grammar, that can be entered as a simple regular expression query in the **Locate** panel:

<N> <N> <N>*

In our previous text, this grammar would recognize the six following sequences:

```
American imports
American imports exports
American imports exports companies
imports exports
imports exports companies
exports companies
```

If the option "Index All matches" is selected, NooJ will produce the six sequences above. If the option "Index Longest matches" is selected, NooJ will produce only the longest: "American imports exports companies".

But if the option "Index shortest matches" is selected, NooJ will produce only the two following results:

```
American imports
exports companies
```

The sequence "imports exports" is not even considered, because after NooJ has matched the sequence "American imports", it simply goes to the right of the matching sequence and continues its parsing.

NooJ produces overlapping matches (and the resulting annotations) only when the option "Index All matches" is selected. With all other options, after NooJ's parser has matched a given sequence (shortest or longest), it simply goes beyond the sequence before applying the grammar again.

Setting modes for grammars that are automatically applied

The user can select grammars in **Info > Preferences > Syntactic Analysis**. In that case, NooJ applies them automatically every time the command **TEXT > Linguistic Analysis** is launched, right after the selected lexical resources.

Each grammar can be applied in sequence; this allows one grammar to insert annotations in the TAS, which will be re-used in turn by other subsequent grammars. For instance, consider the following text:

... from Monday to Thursday ...

a **Date** grammar could be used to annotate the wordforms "Monday" and "Tuesday" as **DATE**; then a **Duration** grammar could be used to annotate sequences such as "from <DATE> to <DATE>" as **DURATION**.

By default, each grammar is applied with the "Longest Match" option; thus, in case of ambiguity, only the longest possible sequences will be annotated. It is possible to override this mode, and tell NooJ to apply a certain grammar in a specific mode. In order to do that, we use a special file name convention. If the suffix of the grammar name is:

-A (e.g. filename: **DET-A.nog**): the grammar will be applied in "All Matches" mode. In other words, in case of ambiguities, the grammar will annotate all matching sequences;

-S (e.g. filename: **NP-S.nog**): the grammar will be applied in "Only Shortest Matches" mode. In other words, in case of ambiguities, the grammar will annotate only the shortest sequences;

-L : (e.g. filename: **Date-L.nog** or **Date.nog**) the grammar will be applied in "Only Longest Matches" mode. In other words, in case of ambiguities, the grammar will annotate only the longest sequences.

Special feature +UNAMB

The special feature **+UNAMB** -- already seen in 12 can be used also by NooJ's syntactic parser. For instance, the following syntactic grammar:

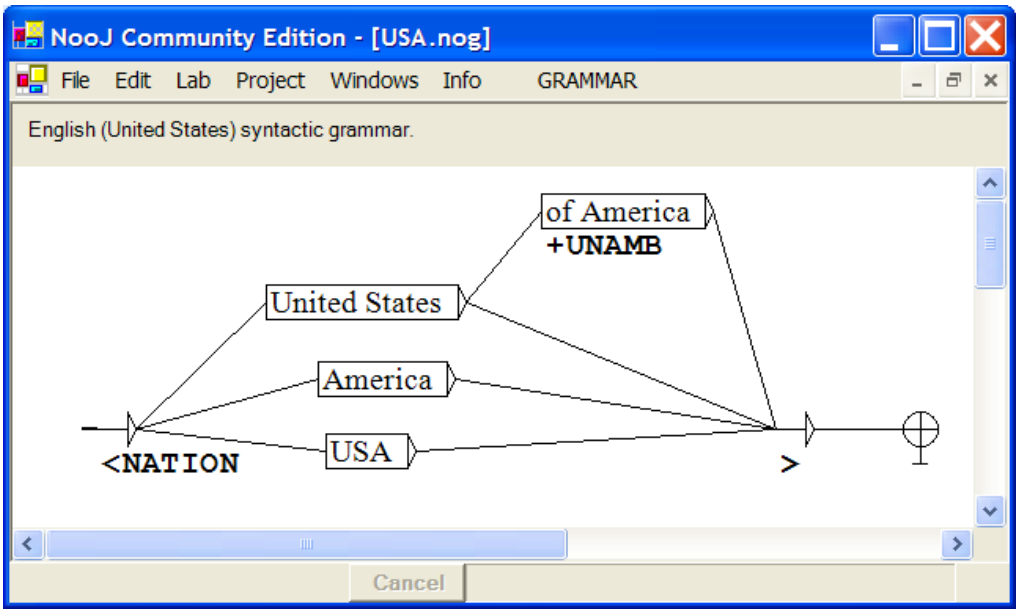


Figure 105. A Syntactic Grammar for USA

recognizes both "United States" and "United States of America". If the longest sequence is recognized, we do not want to parse it as ambiguous. Thus, we add the special feature **+UNAMB** to the annotation by the grammar.

Special feature +EXCLUDE

The special feature **+EXCLUDE** can be used to cancel out a match. For instance, the following syntactic grammar:

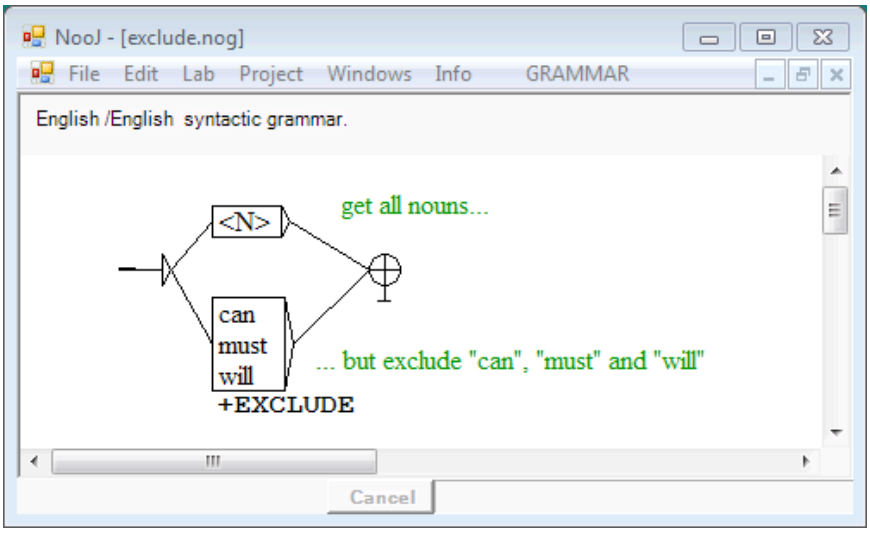


Figure 106. A query with +EXCLUDE

recognizes all nouns except "can", "must" and "will". These three wordforms would match symbol <N> because they might be nouns (e.g. in "a can of coke"), but are mostly verbs. We add the special feature **+EXCLUDE** to exclude these wordforms from the concordance.

Specials features +ONCE and +ONE

The special symbol **+ONCE** is used whenever we want to stop NooJ from following a given path more than once. For instance, in English, date complements are quite moveable in a sentence:

Yesterday John saw a duck. ? John yesterday saw a duck. John saw a duck yesterday.

In consequence, a syntactic grammar that describes full sentences will have to take care of date complements that could occur pretty much between every phrase of the sentences. However, it is not possible to have more than one date complement at the same time:

** Yesterday John saw a duck this morning.*

In order to stop the parser from getting more than one date complement, we can use the **+ONCE** special symbol:

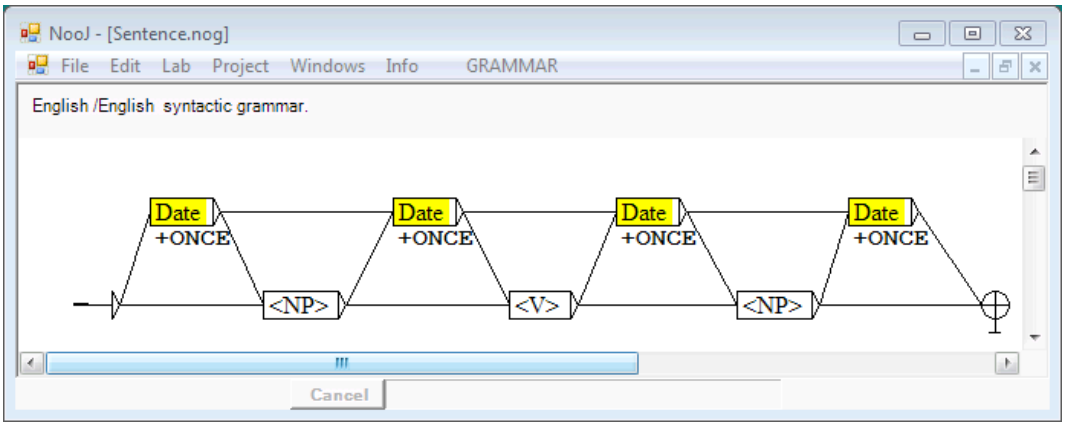


Figure 107. One date complement, anywhere

It is possible to use more than one **+ONCE** constraint at the same time, for instance if one wants to have only one date complement, one location complement, etc. In that case, name each constraint with a different name, e.g.: **+ONCE-DATE**, **+ONCE-LOC,** etc.

Special feature **+ONE** is used to make sure one and only one occurrence of a match occurs during the parsing. This symbol is particularly useful when parsing languages in which words or phrases can occur in any order. For instance, the following grammar could be used to parse some sentences in Latin:

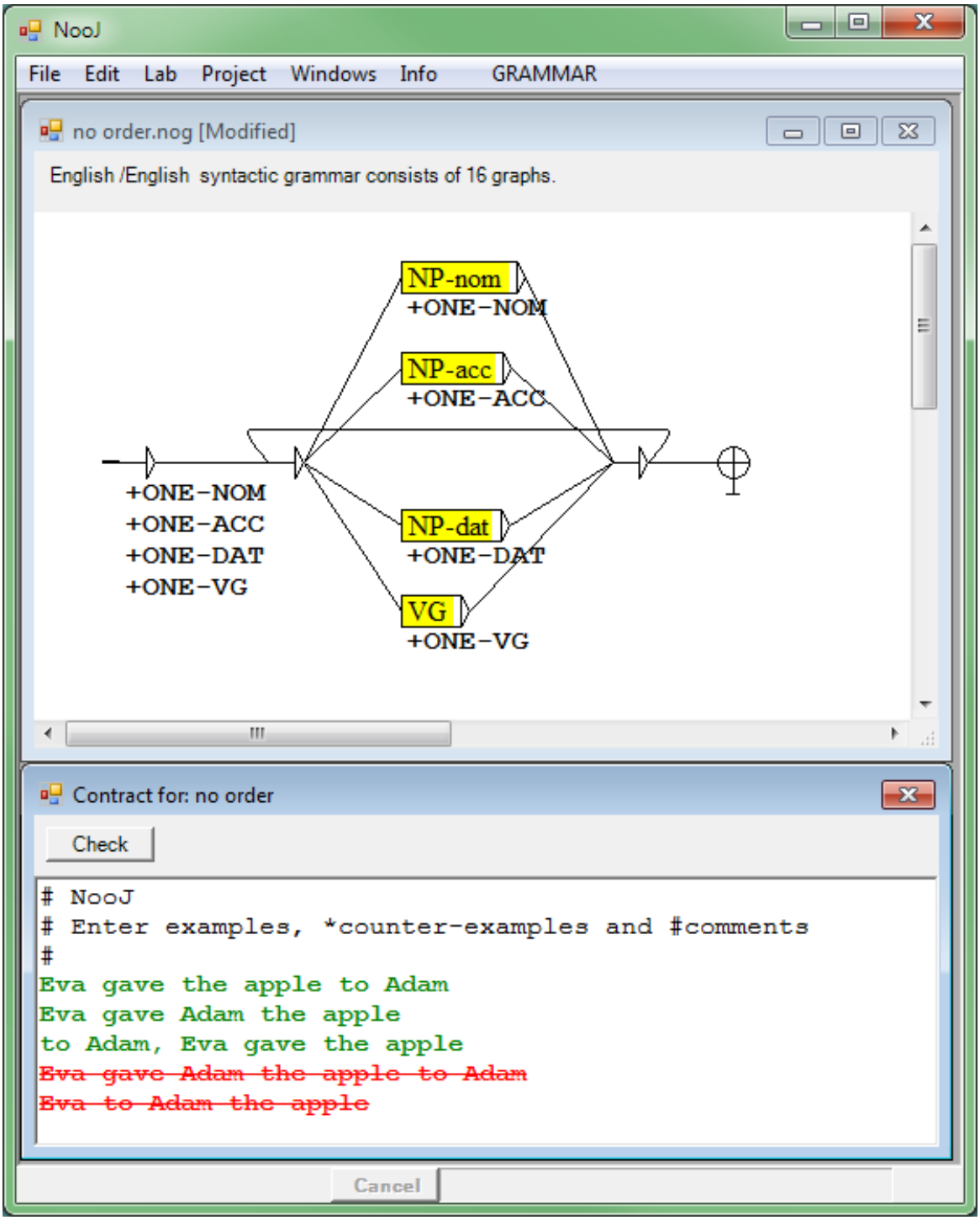


Figure 108. Free noun phrase order

This grammar makes sure that in the resulting analysis, there will be one, and only one nominative noun phrase, one accusative noun phrase, one dative noun phrase and one verb group. This mechanism automatically removes certain ambiguities, such as between nominative and dative nouns: if we know that a certain noun is obligatorily in its nominative form, then all other (possibly ambiguous) nouns will automatically be described as being *not* nominative.

16. Beyond Finite-State Machines

Most grammars we have seen are equivalent to finite-state devices. Although some of them contain several graphs that include references to embedded graphs, such as the **Roman Numeral** or the **Date** grammars, these grammars can be rewritten (or even compiled automatically) into a single, finite-state graph, just by replacing each reference to embedded graphs by the graphs themselves. We now consider more powerful tools that can be used to compute the structure of sentences, to compute paraphrases (i.e. to perform transformations on texts), or to perform machine translation.

Context-Free Grammars

Consider the following grammar:

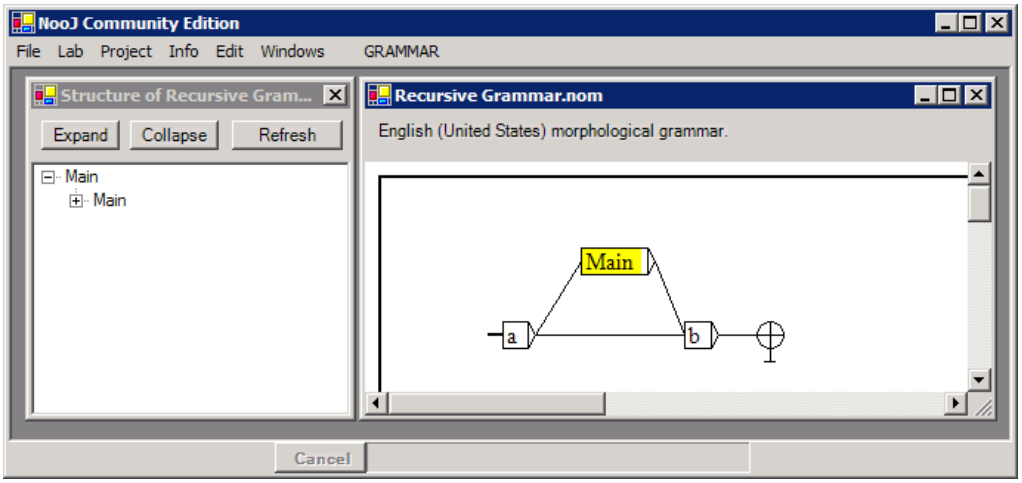


Figure 109. Example of a recursive grammar

In this grammar, the main graph "Main" contains a reference to itself. This graph recognizes the following sequence:

```
a b
```

as well as the sequence:

```
a a b b
```

NooJ's parser matches the first "a", then it recognizes the sequence "a b" thanks to the call to Main in the upper path of the graph; then "b": then, it reaches the terminal node, so the whole sequence is recognized. We can, in the same manner, identify an infinite number of sequences such as:

```
a a a b b b
a a a a b b b b
a a a a a b b b b b
...
```

More generally, this grammar recognizes any sequence composed of a given number of "a", followed by the same number of "b".

It is not possible to "flatten" this grammar into a single graph simply by replacing each reference of an embedded graph by the graph itself. This grammar is said to be a **recursive grammar**. Recursive grammars are also called "Context-Free Grammars", Chomsky's grammars, or Algebraic grammars.

When, in a grammar, a graph contains a reference to itself (as is the case of the graph Main), 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 direct or indirect recursion: *head recursion*, *tail recursion* and *middle recursion*:

Grammars with head recursion

A head recursion means that the recursion is occurring at the left part of the grammar. In other words, the prefix of the grammar contains a direct or indirect recursive call. Consider the following recursive grammar:

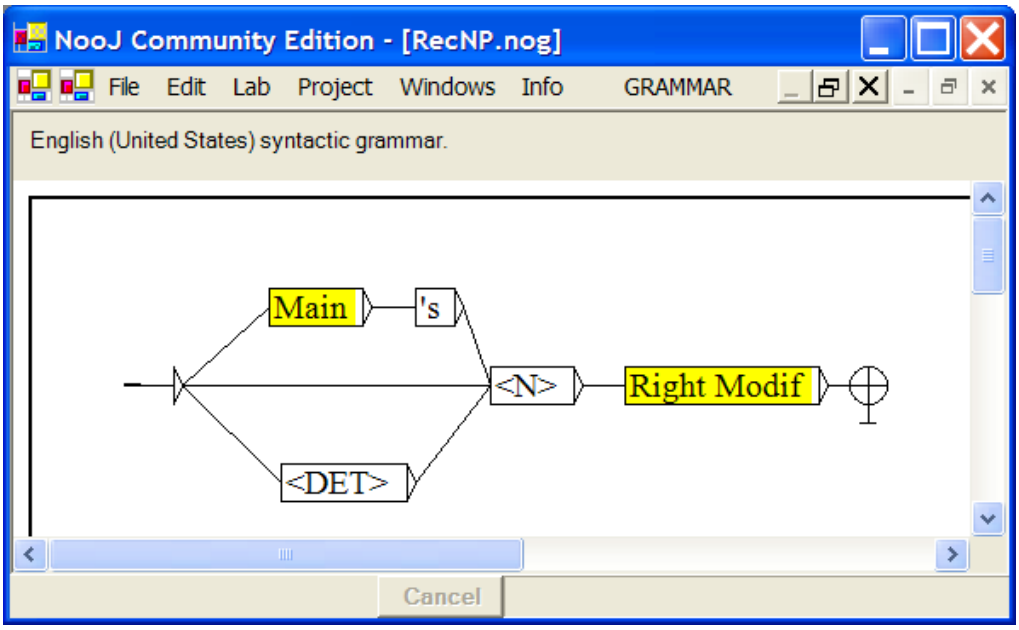


Figure 110. A Grammar with Head Recursion

This grammar represents English noun phrases made up of a determiner, a noun, an optional modifier to the right, as well as noun phrases followed by an "'s" (indicator of the genitive case) and a noun, such as:

```
the neighbour
the neighbour's friend
the neighbour's friend's table
```

This grammar is said to be head-recursive because the reference to the graph Main occurs to the left of the graph Main.

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

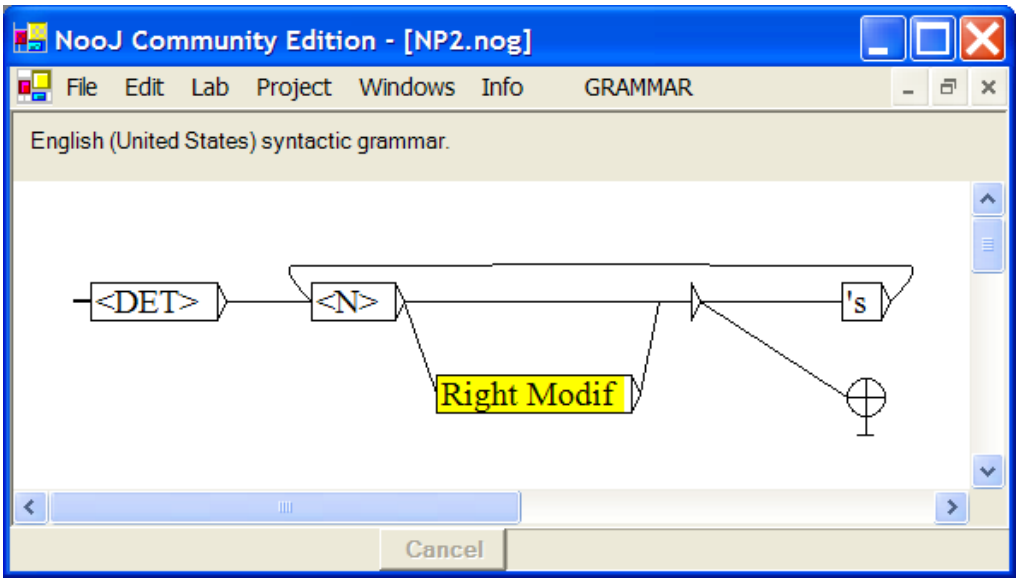


Figure 111. Equivalent grammar, non head recurring

Note: a distributional constraint is not taken into account here: the noun phrase before the "*'s*" must be human (**+Hum**), while the entire noun phrase is not necessarily human. Therefore, writing a recursive call to Main is not correct, and should be replaced with a reference to a different graph that recognizes only **human** noun phrases. This distributional constraint weakens the argument that we need head-recursive grammars to formalize the English syntax.

Grammars with tail recursion

Consider the following grammar:

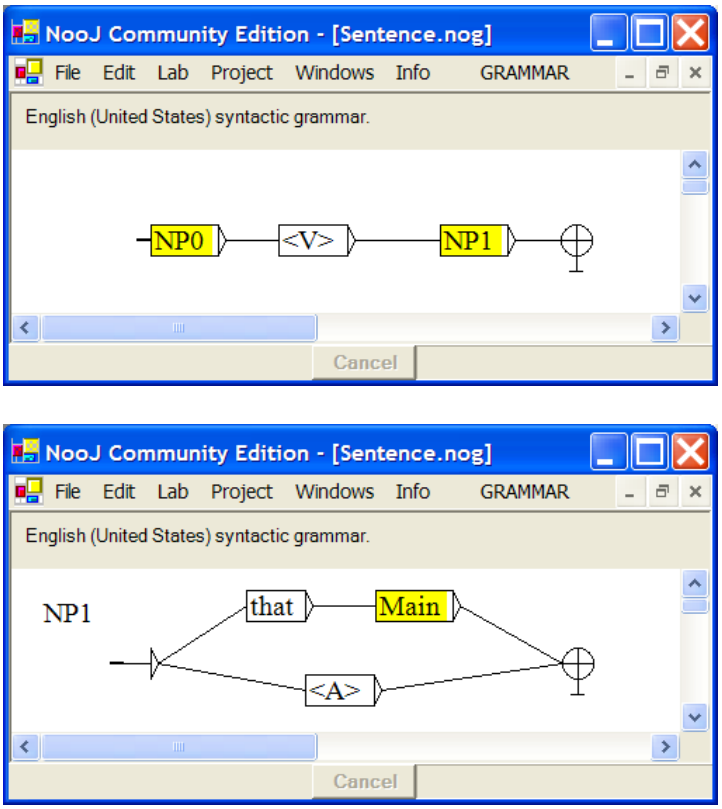


Figure 112. A Grammar with Tail Recursion

This grammar recognizes sequences like the following:

John thinks that Mary said that Peter knows that Lisa declared that Luc is fair

This grammar is indirectly tail-recursive, because the **Main** graph contains a reference to graph **NP1**, which itself contains a reference to the **Main** graph to its right.

There as well, the tail recursion can automatically be removed. For example, the previous grammar **Sentence** is equivalent to the following grammar **Sentence2**, which is a finite-state grammar:

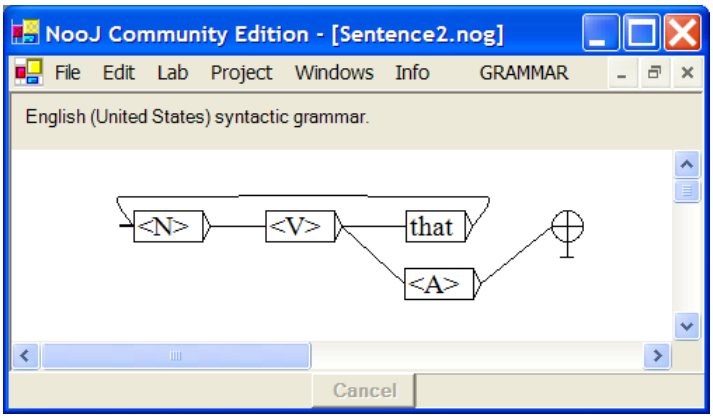


Figure 113. An equivalent finite-state grammar

Note: Here too, there are linguistic constraints that are not taken into account here: each verb needs to be compatible with its corresponding complement **NP1**. Verbs that accept an adjective as complement (such as in "to be fair") should not be processed the same way as verbs that require a sentencial complement (such as "to say that S"). These constraints weaken the argument that we actually need tail-recursive grammars to formalize the English syntax.

Grammars with middle recursion

Contrary to the preceding cases, grammars with a more general recursivity (neither head nor tail), as in the following grammar, are strictly Context-Free, as an equivalent non-recursive grammar cannot be constructed:

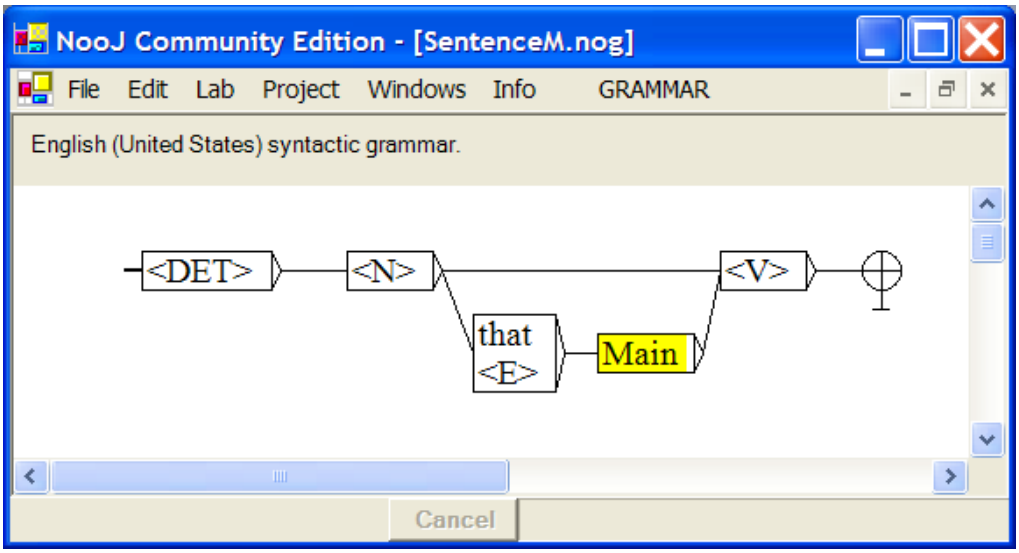


Figure 114. A grammar with middle recursion

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

(1) The cat John bought is white
(2) ? The cat that the neighbour the waiter saw bought is white

It is fairly easy to see that this grammar is a very inadequate description of the syntactic phenomenon that we observe in English. Sentence (2) is already hard to accept, and a further embedded clause would make it unacceptable:

*(3) * The cat that the neighbour the waiter you saw is looking at bought is white*

Moreover, the embedded relative clauses are not complete sentences, as the following impossible construction shows:

** The cat John bought a dog is white*

The relative clause cannot contain a direct object. More precisely, the graph to the right of the relative pronoun should not be **Main**, but rather a different graph that represents **incomplete** sentences, the structure of which depends on the relative pronoun in question.

It is interesting to note that other languages rarely authorize this type of insertions for more than three levels, as we can see in the following French 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, except maybe for some type of tail-recursivity, that natural languages are far less recursive than is generally held.

Context-Sensitive Grammars

Enhanced Grammars are grammars that use internal variables to store parts (affixes) of the recognized sequences, and then use their content in the output, as constraints for the syntactic parser or in order to construct the result of the analysis. This functionality is similar to the way variables are used in programs like SED and PERL. Note that we have already used variables in NooJ's morphological grammars.

In a grammar's output, variable names must be prefixed with the character "$", and must end either with a space, or with the special character "#" when one does not wish to add a space after a variable's content. For instance, the output: "**$Verb ($Name)**" is a valid output that contains references to the two variables **$Verb** and **$Noun**, and might produce the text "**sleep (John)**" (with two spaces); similarly, "**$Verb#($Name#)**" is a valid output that would produce "sleep(John)" (without spaces), whereas "$Verb($Noun)" is an **invalid** output that will produce the error message "**Variable "$Verb(" is undefined**".

Performing transformations

We give now examples of syntactic applications. For example, consider the following grammar:

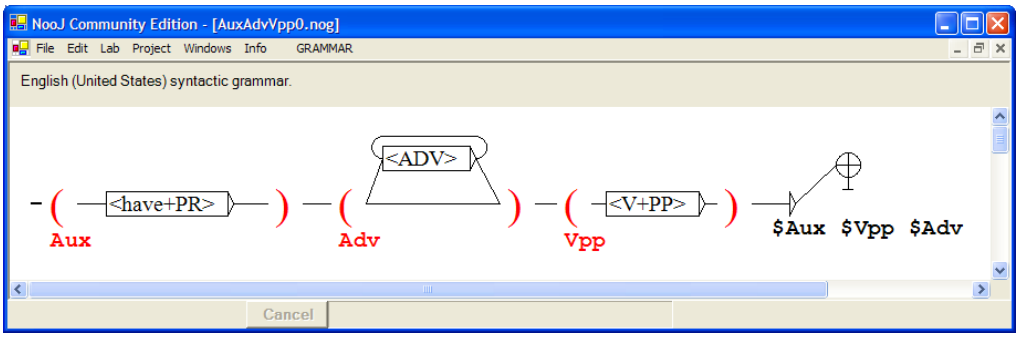


Figure 115. Moving the adverb out of the Verb Group

This grammar could be used to extract the adverbial insertions from the Verb Group, and thus to tag compound tenses and verb constructs as single words. For instance, the sequence "have not yet eaten" could then be annotated as:

```
<have eaten,eat,V+PresentPerfect> <not yet,ADV>
```

To store an affix into a variable, create a node labeled "$(" followed by the variable's name before the text to be recognized, and a node labeled "$)" after the text.

If we apply this grammar to the following text:

My mother has thoroughly mastered the art of condensation

The sequence "*has thoroughly mastered*" is recognized, and the variables $Aux, $Adv and $Vpp are associated to the respective values:

```
$Aux = "has" ; $Adv = "thoroughly" ; $Vpp = "mastered"
```

The result produced by the grammar is then: *has mastered thoroughly*. If we apply the grammar to the text "The portrait of a lady", we get the following concordance:

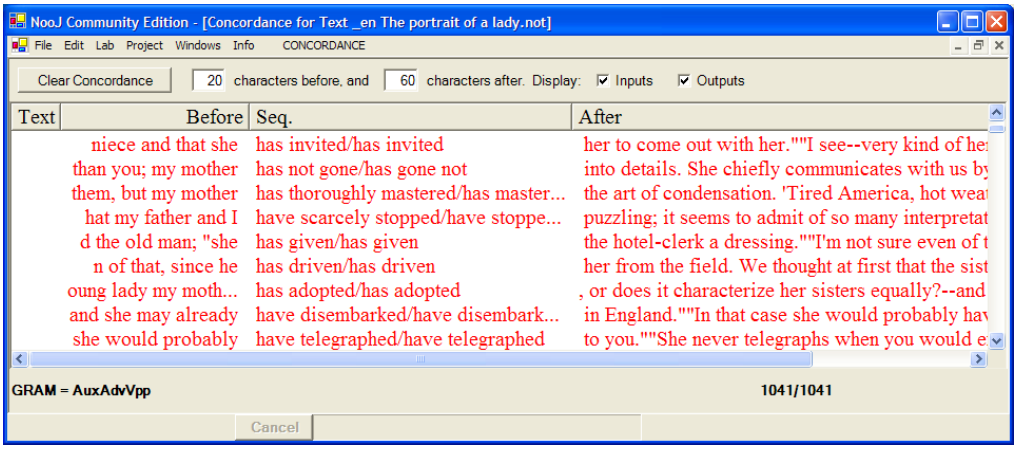


Figure 116. Moving the adverb out of the Verb Group

Override Variables' Value

It is possible to set variables to specific values that are different from the text sequence that was matched by the grammar. In order to do that, enter the variable's value after the variable's closing parenthesis. See for instance the values "M" and "F" in the following grammar:

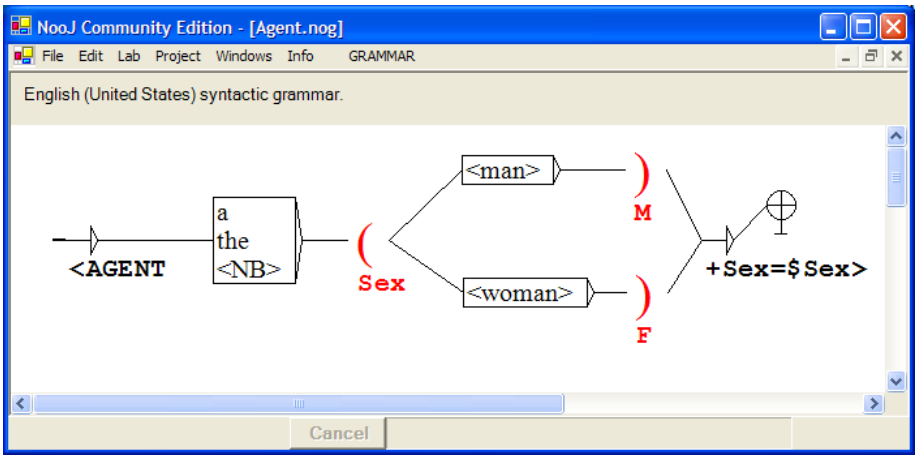


Figure 117. Setting a variable's value

When matching a sequence, the variable **$Sex** is set to either "M" or "F", rather than "man" or "woman". Applying this grammar to a text produces the following concordance:

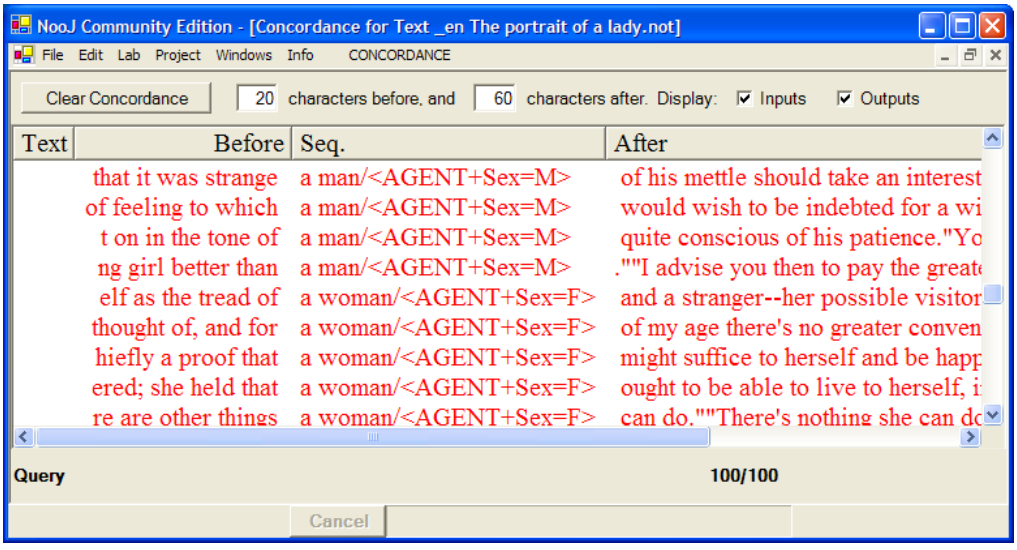


Figure 118. Setting a variable's value

This feature allows users to set values to annotations' properties, that are different from the input text. This can be used to produce abstract representations of the text, such as XML-type or PROLOG-type semantic representations. For instance, the sentence "One woman aged 25 had fever symptom" could be analyzed as:

```
Agent (Sex(F), Age(25), Symptom(fever))
```

or annotated as:

```
<AGENT Sex="F" Age="25" Symptom="fever"> ... </AGENT>
```

Remember to use the special character "#" at the end of the variable's name. For instance, if **$Sex** holds the value "M" and $Age holds the value "25", "**+Sex=$Sex#+Age=$Age#**" will produce "+Sex=M+Age=25", whereas "**+Sex=$Sex+Age=$Age**" will produce the error message "**Variable "$Sex+Age=$Age" is undefined**".

Even at the surface level, it is often useful to be able to render explicit an expression that was either modified or deleted in the original text. For instance, in the following grammar:

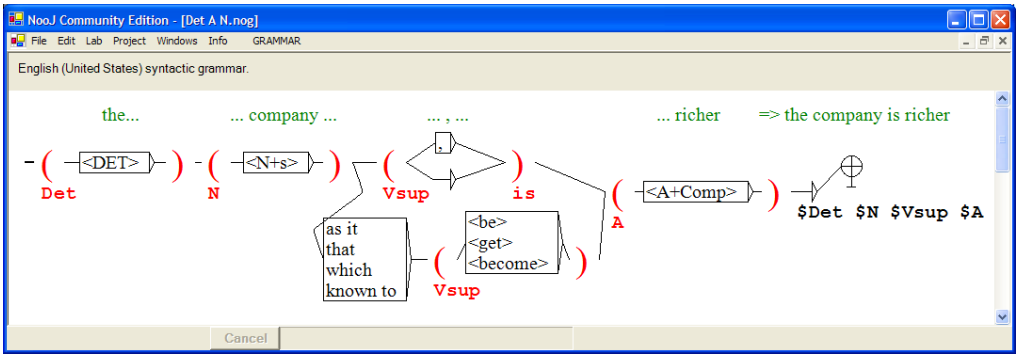


Figure 119. Computing Vsup

the value of variable **$Vsup** is either read explicitely from the text (e.g. to "become" in "*the company known to become richer*"), or is set to "is" if no verb is present (e.g. in "*the company, richer*").

Morphological operations on variables

Variables' content can be processed with NooJ's morphological engine, if the dictionaries NooJ is currently using (i.e. selected in **Info > Preferences > Lexical Analysis**) are providing the corresponding information. For instance, if one of the currently selected lexical resources contains an entry such as:

```
eat,V+FLX=EAT+tr+...
```

where the paradigm **+FLX=EAT** allows NooJ to recognize the wordform "eaten" as a valid inflected form for "eat", then NooJ can lemmatize "eaten" into "eat".

Lemmatization operator: in order to lemmatize the content of a variable, use the operator "_" (underline character) to the right of the variable's name.

For instance, "**$V_**" takes the content of variable **$V**, e.g. "ate", and then lemmatizes it into "eat".

Similarly, if the conjugation paradigm for the verb "eat" associates the inflectional code "**+PP**" for its Past Participle form "eaten", then the expression "**$V_PP**" will produce the wordform "eaten" from any other wordform of "eat".

Remember to use the special character "#" at the end of the variable's name when composing a grammar's complex output. For instance, if **$V** holds the value "distributes", "**$V_**" will produce "distribute", whereas "**$V#_**" will produce "distributes_".

Inflection: in order to inflect the content of a variable, use the operator "_" (underline character) followed by one or more inflectional codes that characterize the resulting inflected form.

Another example: the expression "**$N_p**" takes the content of variable **$N**, e.g. "woman", then inflects it in the plural, e.g. "women"; the expression "**$V_PR+3+s**" takes the content of variable **$V**, e.g. "ate", and then inflects it to Present, third person, singular, e.g. "eats".

Derivational operations on variables

Variables' content can be processed also with NooJ's derivational morphological engine, if the dictionaries NooJ is currently using (i.e. selected in **Info > Preferences > Lexical Analysis**) are providing the corresponding information. For instance, if one of the currently selected lexical resources contains an entry such as:

```
distribute,V+DRV=TION+...
```

where the paradigm **+DRV=TION** links the lexical entry "distribute" to the derived form "distribution", then one can nominalize "distribute" into "distribution" by using the expression "**$V_N**".

Derivation: in order to derive the content of a variable, use the operator "_" (underline character) followed by a category and one or more inflectional codes, to the right of the variable's name.

A derivation operator can be followed by an inflection operator. For instance, "**$V_N+p**" takes the content of variable **$V**, e.g. "distributed", then nominalizes it (e.g. "distribution"), and computes the plural form of the result, e.g. "distributions".

Extracting Variables' Lexical Properties

NooJ can retrieve and extract values of a property that is associated with the variable's content. For instance, if one of the currently selected lexical resources contains an entry such as:

```
pen,N+Class=Conc+...
```

then NooJ can extract the value "Conc" from the lexical entry's property name "**Class**".

Property: in order to extract the value of a variable's property, use the name of the property to the right of the variable's name.

For instance, if the variable **$Noun** holds the lexical entry for "pen", then the output **$Noun$Class** will produce the text "Conc".

Remember to use the special character "#" at the end of the compound variable's name. For instance, **$Noun$Class** refers to the property "Class" of the variable "$Noun", whereas **$Noun#$Class** will concatenate the content of the two different variables **$Noun** and **$Class**.

Of course, retrieving a lexical entry's property value requires that the lexical property be expressed in terms of a attribute-value pair, such as in the previous example: **+Class=Conc**. But NooJ lexical properties can also be expressed as single features such as "**+Present**", "**+Hum**" or "**+s**", as in:

```
teacher,N+Hum+s
```

In order to extract simple features from a lexical entry, we need to use NooJ's "properties.def" property definition file. For example, consider the following definitions stored in a "properties.def" definition file:

```
N_Class = Hum + Conc + Anim + Abst ;
N_Nb = s + p ;
```

These definitions state that "**Class**" is valid property for category "**N**" (Nouns); it can take one of the following values: **Hum**, **Conc**, **Anim** and **Abst**. "**Nb**" is a valid property for Nouns; it can take the two values "**s**" (singular) or "**p**" (plural).

If these two rules are written in the current "properties.def" file, then the previous lexical entry is exactly equivalent to the following one:

```
teacher,N+Class=Hum+Nb=s
```

Then, it is possible to access the features "Hum" and "s" by using variables' property queries such as **NClass** or **NNb**.

Lexical and Agreement Constraints

Just like morphological grammars, syntactic grammars can contain lexical and agreement constraints. Constraints allow grammar writers to separate the "constituent" part of a grammar from the "agreement" constraints, thus simplifying grammars considerably. For instance, when describing French noun phrases, one can write a simple grammar such as:

<DET> <N> <A>

and associate it with the two following agreement constraints:

<DETNb=NNb>, <ANb=NNb>

Applications

Transformational Analysis

Being able to use variables and to perform morphological operations on them allows us to perform the four basic transformation operations:

-- delete elements: for example, the substitution: **$1 $2 $3 / $1 $3** allows us to erase the sequence stored in variable **$2** in the left context of **$1** and in the right context of **$3**.

This feature is useful when erasing certain grammatical words, such as in:

The table that John bought = The table John bought

-- insert elements: for example, the substitution: **$1 $2 / $1 XXX $2** allows us to insert the text **XXX** between sequences **$1** and **$2**.

This feature can be used to analyze certain types of support verb deletions; for instance, the sentence "the red table" can be automatically linked to "the table which is red", where the verb "is" is made explicit.

-- dupplications: 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 text.

This feature can be used to analyze certain types of coordinations; for instance, the sentence "The red tables and chairs" can be automatically linked to "The red tables and the red chairs", where the modifyer "red" is duplicated.

-- permutations: for example, the substitution : **$1 $2 $3 $4 $5 / $1 $4 $3 $2 $5** allows us to swap the respective positions of **$2** and **$4**.

This feature can be used to perform sentences' transformations. For instance, the sentence "John ate 3 apples" can be automatically linked to "3 apples were eaten by John", where the subject "John" and the object "3 apples" are permuted, and the verb "ate" is replaced with "were eaten" thanks to the output "were $V_PP".

The use of such transducers in sequence gives NooJ the power of a Turing machine, and makes it a unique tool to perform automatic transformational analyses of texts.

Semantic Analyses

NooJ grammars can be used to recognize certain sequences and produce a corresponding semantic analysis, expressed in any formalism, including XML or PROLOG-like predicate-argument constructions.

For instance, consider the following grammar:

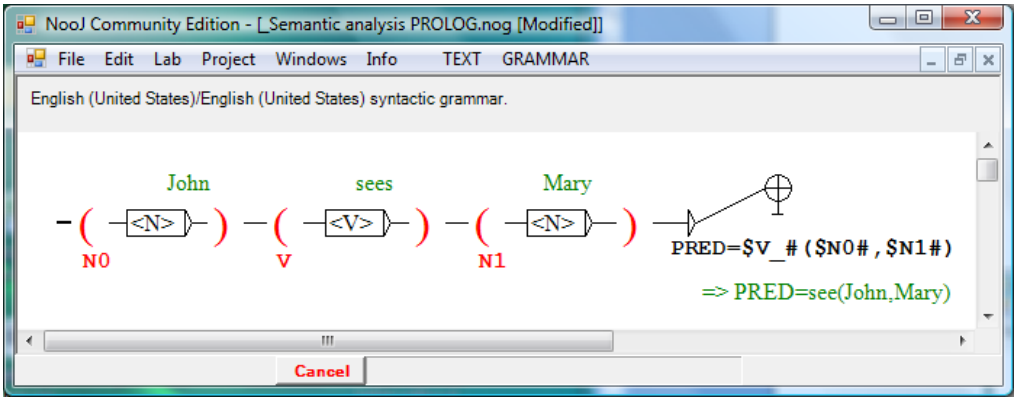


Figure 120. A Prolog Semantic Analysis

This grammar recognizes sequences such as:

John sees Mary

and then, produces the corresponding PROLOG-like analysis:

```
PRED=see (John,Mary)
```

Of course, the formalism used to format the output is totally open. For instance, the previous grammar can be rewritten to produce an XML-like analysis:

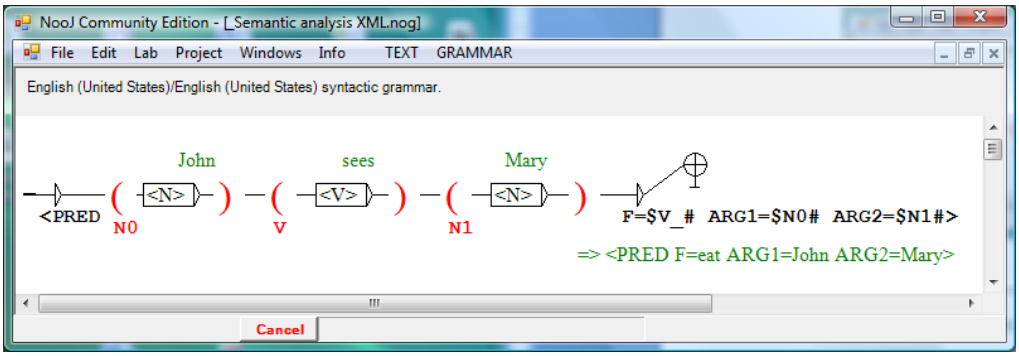


Figure 121. An XML Semantic Analysis

In that case, the same sequence "John sees Mary" will be assocated with the output:

```
<PRED F="see" ARG1="John" ARG2="Mary">John sees Mary</PRED>
```

Machine Translation using multilingual dictionaries and grammars

NooJ dictionaries are associated with one single input language (the one that has to be specified when creating a new dictionary via the command **File > New Dictionary**). All lexical entries of the dictionary must belong to this language.

However, lexical entries can be associated with properties which can hold values in other languages than the dictionary's. For instance, consider the following English lexical entry:

```
pen,N+Conc+FR="crayon"+IT="penna"+SP="pluma"
```

The properties' names "FR", "IT" and "SP" refer to language names. Therefore, if a grammar variable, say **$Noun**, holds the value "pen", the output **$Noun$FR** will produce the text "crayon".

Now consider the following dictionary entries:

```
Monday,N+FR=lundi
Tuesday,N+FR=mardi
Wednesday,N+FR=mercredi
Thursday,N+FR=jeudi
Friday,N+FR=vendredi
Saturday,N+FR=samedi
Sunday,N+FR=dimanche
```

and the following grammar:

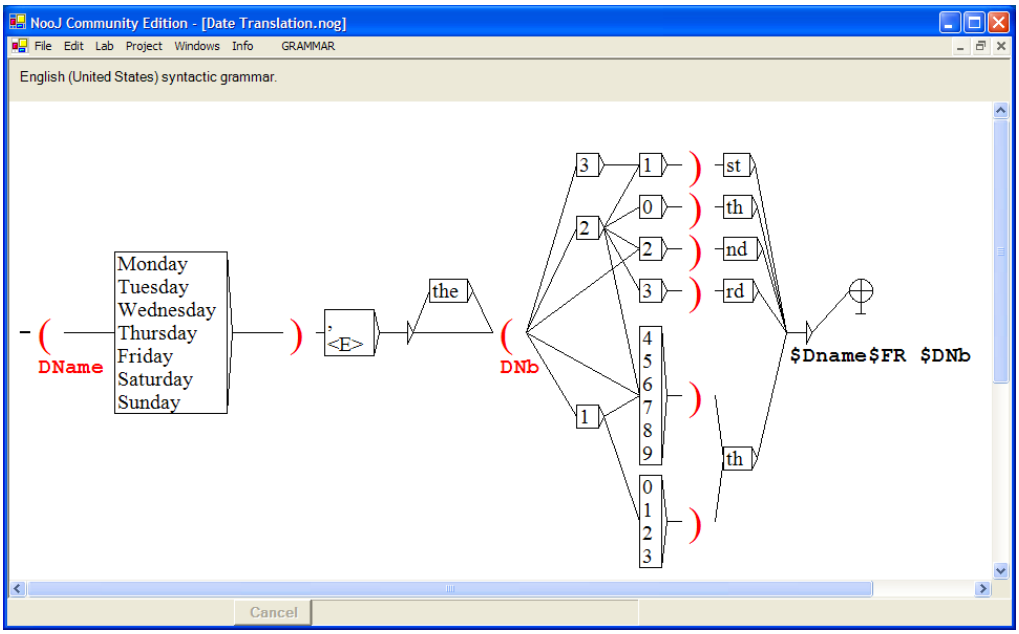


Figure 122. A simple translation

The output "**$DName$FR $DNb**" takes the name of the day, translates it into French, and then adds to it the date number. For instance, "Friday, the 13th" and "Friday 13th" will be translated into "vendredi 13".

Exporting results and the TRANS operator

See p.71

REFERENCES

Chapter 17 presents three important NooJ functionalities: Labs, Projects and Testing. These tools allow teachers to rapidly design and implement experiments that can be shared with students and other research partners and developers to check the consistency of their resources. Chapter 18 lists and describes all NooJ commands. Chapter 19 describes NooJ's command-line programs and Object Oriented Interface (API). Chapter 20 presents bibliographical references.

17. Development tools

NooJ Labs

NooJ Labs are small applications that are designed either to demonstrate a specific linguistic phenomenon and how to represent it with NooJ, or to provide users with a specific, useful tool for NooJ. All labs are available from NooJ's menu item "**Lab**".

As of now, there are six available labs, but it should be easy to add more; NooJ is based on an "Object-Oriented Component Programming" technology (.NET Framework), which allows programmers to add modules to their application very easily. We hope that people who use NooJ will design new labs: contact us if you wish to add a lab.

Lab > Text Encoding

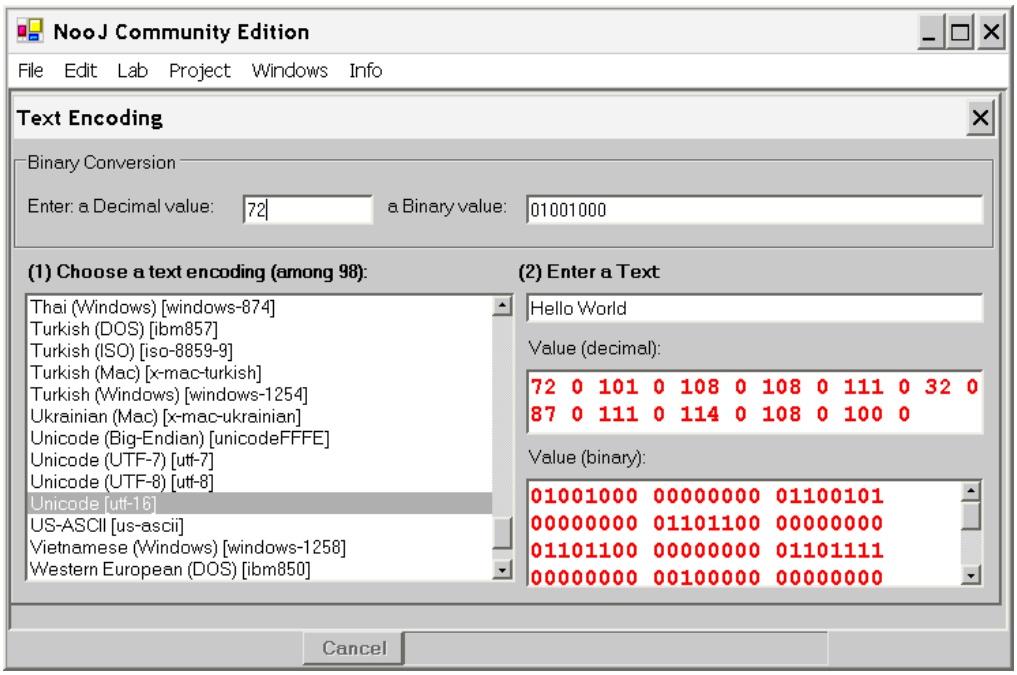


Figure 123. Lab > Text Encoding

This lab is designed to demonstrate how texts are represented in computer files.

The top of the window shows the correspondance between the decimal and the binary notations for numbers. Students can enter a value in decimal (e.g. "72"), then press "Enter" to get the number written in binary form, or they can enter a number in a binary form, then press "Enter" to display the number in decimal form.

Students should learn how to translate a number from decimal to binary, and from binary to decimal.

The bottom of the window shows that a text file is represented by a series of numbers, each number corresponds to each character of the text. Note that, depending on the selected encoding format, each character is represented (i.e. "encoded") by:

-- one single number which takes its values from 0 to 128 (e.g. ASCII),

-- one number from 0 to 255 (e.g. the various extensions for ASCII-8),

-- two numbers from 0 to 255 (e.g. Unicode 16),

-- by a code of variable length; e.g. Unicode 8 represents the letter "a" as the single number "97" and the accented letter "à" as "197, 160".

The character codes are displayed both in decimal and in binary.

Students should enter texts that contain lowercase and uppercase letters, accented letters, non-roman characters (e.g. Chinese), special characters (e.g. space and

tabulation) and special cases (e.g. a given letter both in final and in non final mode), etc.

Students should see how different character encoding standards are more or less compatible with each others (for instance the different variants of ASCII), as well as the limitations of some character encoding standards (e.g. ASCII).

Lab > PERL

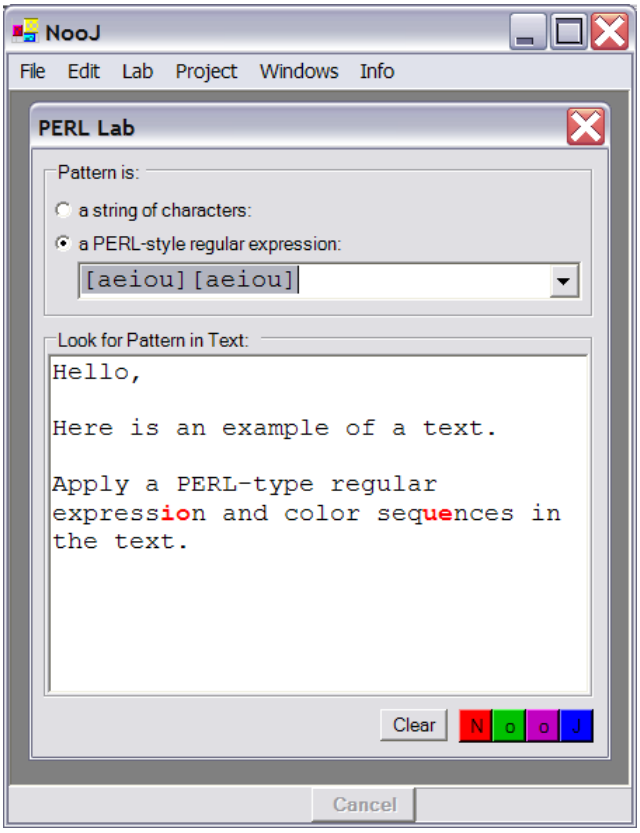


Figure 124. Lab > PERL

This lab is designed to demonstrate how to use PERL-type regular expressions to express complex queries on string of characters.

In NooJ, PERL-type regular expressions are used in the Dictionary window: it allows users to perform complex search, replace and counting operations on dictionaries.

PERL-type regular expressions are also used in NooJ lexical symbols (see 0PERL-type , p. 49). It allows users to construct complex morpho-syntactic queries, such as for instance **<ADV-MP="lly$">** (to get all the adverbs in the text that do not end with "lly").

The two options at the top of the window allow users to look for a pattern in the text: either a simple string of characters, or a PERL-type regular expression.

Users can enter a small text, and then color this text by entering a query. Multiple queries may be launched: each query's results will then be colored-coded.

Students should be invited to enter a small text, then to apply several PERL-type expressions.

Lab > Construct A Corpus

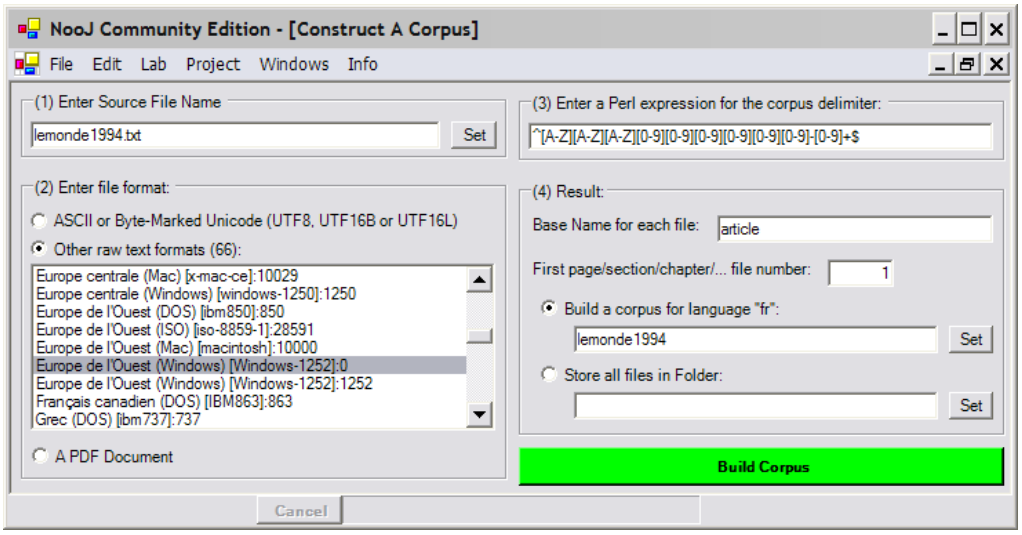


Figure 125. Lab > Construct A Corpus

This lab allows users to split a potentially large text file into a series of small text files, and store them into a NooJ corpus file.

(1) Enter the source file name

(2) Enter the character encoding format for the source file. Make sure to select the right encoding for your file. For instance, the default encoding on my Windows PC machine is **not** "ASCII or Byte-marked Unicode", but rather "Windows Western European-1252".

(3) Enter a description of the delimiter that will be used to split the source file. For instance, if the original source file contains a series of articles that are delimited by a special line such as "=====", just enter this character string.

Beside pure character strings, NooJ allows users to specify a delimiter by entering a PERL-like expression. For instance, if each article starts with a code that consists of three uppercase letters, followed by six-digit date, followed by a dash, followed by one or more digits, e.g.:

DEC011294-32

...

DEC011294-33

...

DEC011294-34

...

then a proper description for these delimiters could be:

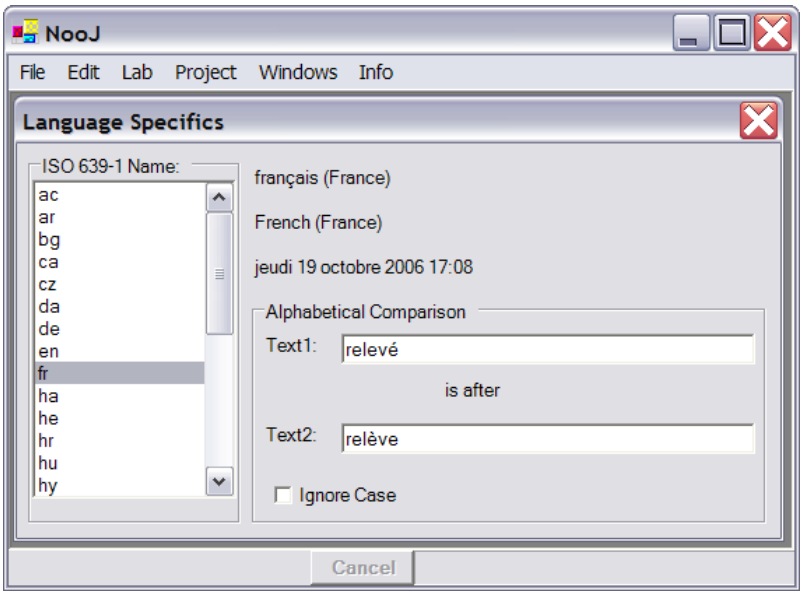
```
^[A-Z][A-Z][A-Z][0-9][0-9][0-9][0-9][0-9][0-9]-[0-9]+$
```

(PERL-type expressions are well documented, and numerous tutorials are available on the Internet).

When processing a PDF document[8], NooJ splits the document into a series of pages. In other words, if the PDF document is a 300-page book, the resulting corpus will contain 300 text files. Parsing a page-based corpus with NooJ allows users to index a book very easily: a NooJ concordance for terms or proper names will display the page number for each match.

(4) The result is either a NooJ corpus, or a set of individual text files. In the first case, make sure that the language for the resulting corpus is right. To change it, go to Info > Preferences.

Lab > Language Specifics



[8] In order for NooJ to process non-encrypted PDF files, one needs to install the free software pdf2t2xt.exe in NooJ's _App folder.

This lab is designed to present each language's alphabetical order. Select a language, and then type in two words: NooJ will apply its comparison routine according to the current language to sort the two words.

Alphabetical orders in various languages are not straight forward. For instance, in French, comparing two wordforms involves two steps: first, we take out accents and diacritics; if the two words (without diacritics) are identical, then we put back the diacritics and we compare the two wordforms **from right to left**. For instance, the wordform "relevé" is **after** the wordform "relève", because at the second step, the final "é" of relevé is after the final "e" of relève.

NooJ Projects

Projects are packages of NooJ files: texts, dictionaries, grammars, etc. that are associated with some documentation. Projects allow teachers to present or demonstrate a specific problem to a large number of students, without having to deal with complicated setups: when a number of students run a project, they all have exactly the same things on the screen, and exactly the same linguistic settings.

NooJ includes a few English projects:

en Date Translation en2fr: demonstrates how to translate English dates to French

en Export Annotations: demonstrates how to export a NooJ text or corpus into an XML document

en Lexical Analysis: demonstrates how NooJ uses dictionaries and morphological grammars to perform lexical parsing

en Passive: demonstrates how to perform a Passive transformation

en sing2plur: demonstrates how to perform a Singular to Plural transformation

To run a project, click **Projects > Run Projects**.

It is very easy to build your own projects: just open all the files you wish to store in the project, make sure to save them, then click **Projects > Save Configuration As Project**.

A project's window is displayed: it displays all the parameters of the configuration (such as the linguistic resources that are loaded), and allows you to enter a documentation. At this point, you can still modify the configuration, by closing windows that you don't need, or loading other resources (when you are ready, don't forget to update the configuration by clicking **REFRESH**).

To save the project, just click **SAVE**. When you are done, you can close it, and then run it (**Project > Run**) for a check.

Always make sure you close the current project when you are done with it. When NooJ runs in **Project mode**, several functionalities are either disabled (such as the possibility to perform a linguistic analysis of the text) or work differently that usually (the default Preferences are usually different than the ones saved in a Project).

NooJ Automated Testing

NooJ allows developers to automatically test their configuration and linguistic resources against a list of reference test files.

Every time one linguistic resource is modified in NooJ, the users run the risk to destroy the analysis of their texts. For instance, if one removes a lexical entry from a dictionary (i.e. to get rid of an annoying ambiguity), then it is possible that a certain grammar will no longer work, or another text will not get a correct analysis or concordance. In order to contain this problem, NooJ offers the possibility to run a series of tests that check the consistency of the linguistic resources.

(1) Place a number of reference texts (.not files) in the "Tests" folder, inside the MyDocuments/NooJ folder.

(2) Apply the command "Projects > Run Tests".

NooJ will automatically perform a linguistic analysis of each text (.not file) present in the "Tests" folder, and compare the results with the original results. The tests is displayed in the Console window. If an analysis is not identical to its corresponding reference, NooJ will produce a "Fail" error message.

18. NooJ's command line programs and API

Set up the PATH environment variable

Before using NooJ programs directly from a DOS or a UNIX command-line, the complete NooJ package must have been installed in a directory. There is only one package (NooJ.zip) for Windows, LINUX, MAC-OSX & UNIX packages. This file must be uncompressed, and its content's file structure must be preserved.

The main difference between the different OS is that on Windows, NooJ as well as all its command-line programs run on top of ".NET Framework"; this framework is not available on other platforms. However, the corresponding platform "MONO" can be used on the other OS to run the command-line programs. See: www.mono-project.com for more information on MONO.

In order for your OS to find NooJ command-line programs, you will need to set the **PATH** environment variable:

Let's say that you installed NooJ at the top of the "C:" hard disk directory hierarchy. Here is the command that you can add to the file "autoexec.bat" in order to run NooJ command-line programs from any directory in a DOS/Windows environment:

```
@set PATH=%PATH%;c:\NooJ\_App
```

Here is the line to be added to the file "profile.ksh" in order to run NooJ command-line programs from any directory in a Windows/Bourne Shell environment (I use **cygwin**):

```
export PATH="$PATH;/C/NooJ/_App"
```

Here is the line to be added to the file ".profile" in order to run NooJ command-line programs from any directory in a UNIX/LINUX/Bourne Shell environment:

```
export PATH="$PATH;/NooJ/_App"
```

Application and Private directories

Make sure you do not overlook the difference between NooJ's **private** and **application** directories:

The application directory is the directory where you installed the NooJ application; it contains all NooJ programs and DLLs, as well as the original dictionaries and grammars included in NooJ. This directory and all its sub-directories should be write-protected, so that no user could accidentally destroy or modify any of their files.

On a Windows machine, NooJ could be installed in the directory "c:\", or more typically, in the directory "c:\Program files"; in that case, the NooJ application's directories would be the following:

```
c:\Program files\NooJ
c:\Program files\NooJ\_App
c:\Program files\NooJ\_Projects
c:\Program files\NooJ\en
c:\Program files\NooJ\fr
```

Each language directory in turn has three sub-directories, e.g.:

```
c:\Program files\NooJ\en\Lexical Analysis
c:\Program files\NooJ\en\Syntactic Analysis
c:\Program files\NooJ\en\Projects
```

Never, ever, edit, modify or delete the content of any of these application directories. If you add, edit or modify a file in one of these directories, you will loose it at the next upgrade.

Usually, each Windows or UNIX/LINUX/MacOSX user on a computer 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.

On a Windows PC, the private directory is usually stored in "My documents", e.g.: "My Documents\NooJ". In that case, the following directories are created for each user:

```
My Documents\NooJ\en\Lexical Analysis
```

My Documents\NooJ\en\Syntactic Analysis
My Documents\NooJ\en\Projects
My Documents\NooJ\fr\Lexical Analysis
My Documents\NooJ\fr\Syntactic Analysis
My Documents\NooJ\fr\Projects

NooJ users typically add their own texts, corpora, dictionaries and grammars in NooJ's private directories.

You may modify texts, corpora, dictionaries and grammars that belong to the NooJ package. However, in that case, make sure to rename the modified files so that their names does not start with the special character "_" (underline), because when you perform an upgrade, all "_" resources might be updated or erased.

Command-line program: noojapply

The following commands correspond to the "noojapply.exe" file that is stored in the application directory **NooJ_App**.

1. noojapply languagename dictionary

This command compiles a NooJ dictionary ".dic" file to produce a ".nod" file. For instance:

noojapply en mydictionary.dic

produces the file **mydictionary.nod**.

The dictionary might contain #use commands to associate its entries with one or more inflectional/derivational description files ".nof" files. In that case, all associated files must be stored in the same directory as the dictionary.

2. noojapply languagename lex-resources texts

This command applies a set of dictionaries and morphological grammars to a series one ore more texts. For instance:

noojapply en bactery.nod virus.nod ization.nom text1.txt text2.txt

applies the given dictionaries (.nod files) and morphological grammars (.nom files) to each text (.txt file), and then produce the index of all recognized terms.

WARNING: all .txt text files must be encoded as valid Unicode UTF-8 files.

Dictionaries and morphological grammars are applied one after each other, i.e. the first argument has priority 1, the second one has priority 2, etc.

Note INTEX users: any NooJ dictionary can contain both simple words as well as multi-word units, and NooJ (as opposed to INTEX) processes their inflection. Therefore, **noojapply** replaces the two INTEX programs: **dics.exe** and **dicc.exe**, and processes DELAS / DELAC / DELAV dictionaries (no more need for DELAFs or DELACFs).

3. noojapply languagename lex-resources syn-resources texts NOT

applies the given lexical resources (.nod and .nom files) and then syntactic resources (.nog files) to each text (.txt file), annotates them, and then produce each resulting .not (NooJ annotated) text file. For instance:

noojapply en sdic.nod date.nog text.txt NOT

applies **sdic.nod** and **date.nog** to **text.txt**, annotates it, and then saves the result as **text.not**.

Dictionaries, morphological grammars and syntactic grammars are applied one after each other, i.e. the first argument has priority 1, the second one has priority 2, etc.

4. noojapply languagename lex-resources syn-resources query texts

applies a query to one or more unprocessed .txt text files.

All the lexical and syntactic resources are applied before the query. In consequence, the query can contain symbols (e.g. <ADV> or <DATE>) that relate to annotations that have been inserted by a lexical or syntactic resource (e.g. sdic.nod or date.nog).

The query can be in the form of a syntactic regular expression (stored in a ".nox" text file), or in the form of a syntactic grammar (a ".nog" file).

WARNING: all .nox query files must be encoded as valid Unicode UTF-8 files. The resulting index is always encoded as a UTF-8 text file.

For instance, the following command:

noojapply en sdic.nod query.nox text1.txt text2.txt

applies the syntactic grammar **query.nox** to the two text files, and then produces the index of all sequences recognized by the grammar **query.nox**.

noojapply en sdic.nod date.nog query.nog text.txt

applies the lexical resource **sdic.nod** and the syntactic resource **date.nog** before applying the syntactic grammar **query.nog** to the text. NooJ produces the index of all sequences recognized by the grammar **query.nog**.

If the query produces outputs, the resulting index will contain not only the location of all matching sequences, but also the corresponding outputs (typically: their analysis or their translation).

Note INTEX users: NooJ process syntactic grammars directly, i.e. it performs their compilation during run-time (as opposed to INTEX, which needs to compile grammars into Finite-State Transducers); moreover, NooJ decides when to use (or not) indices of the texts to speed up the parsing. Therefore, **noojapply** replaces the four INTEX programs: **recon.exe**, **recor.exe** , **reconind.exe** and **recorind.exe**.

It is possible to apply a number of lexical resources as well as a number of syntactic resources before applying a query to a series of texts. When more than one grammar is given as an argument, the last grammar is used as the query, while the preceeding ones are used as syntactic resources (to annotate the text). For instance, the following command:

noojapply en sdic.nod synt1.nog synt2.nog synt3.nog text1.txt text2.txt

will apply to **text1.txt** and **text2.txt** the dictionary **sdic**, and then the two syntactic grammars **synt1.nod** and **synt2.nod** in cascade, before finally applying the query **synt3.nog**. The result will be the index of all the sequences that match the query **synt3.nog**.

Applying several grammars in cascade allows a user to incrementally annotate the text: each syntactic grammar can then use the annotations that were added by the preceeding grammars.

Note INTEX users: because NooJ applies a number of syntactic grammars in cascade without having to modify (i.e. destroy) the original text, **noojapply** replaces and simplifies the use of the two INTEX programs: **fst2txt.exe** and **highlight.exe** that were used to perform cascading analyses.

5. noojapply languagename query textorcorpus

applies a query to one NooJ annotated .not text file or to one NooJ corpus .noc file. For instance:

noojapply en date.nog promed259.not

will locate all dates in the annotated text promed259.not, and:

noojapply en date.nog promed.noc

will locate all dates in each of the texts that constitute the promed.noc corpus file.

19. Bibliographical References

The NooJ project is the result of numerous research works both in linguistics, computational linguistics, corpus linguistics and in computer science; it would be impossible to construct a complete reference list for these domains. Therefore, we choose to provide below references that are relevant to the INTEX software and linguistic resources (which remain largely compatible with NooJ), and then references to new aspects of NooJ's software, linguistic approach and resources.

Background: INTEX

The following provides a basic reference to the numerous projects that are either related to the construction of INTEX (the previous version of NooJ, which NooJ is largely compatible with), to INTEX modules for 20+ languages, the use of INTEX to process corpora or to perform extract information from various types of texts (literary, technical and journalistic), or to the use of NooJ as a development environment used to implement specific functionalities in NLP computer applications.

Books

Although the following book lists studies that are not directly related to INTEX or NooJ, we feel that it is a very useful source for linguists and NooJ users:

The following book presents the French DELA database which was used by INTEX. Although NooJ's system of dictionaries is an improvement on the DELA, the book discusses issues that are still relevant to NooJ users that wish to formalize a new vocabulary.

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 Eds. 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 book presents various aspects and functionalities of the INTEX program:

Silberztein Max. 1993. *Dictionnaires électroniques et analyse automatique de textes. Le système INTEX*. Masson: Paris (240 p.).

Proceedings of 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.

Muller Claude, Royauté Jean, Silberztein Max Eds. 2004. *INTEX pour la Linguistique et le Traitement Automatique des Langues*. Proceedings of the 4th and 5th INTEX workshops, Bordeaux, May 2001, Marseille, May 2002: Presses Universitaires de Franche-Comté (400 p).

Various papers (not in the proceedings above)

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. 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. NooJ: a corpus processing system, in *COLING 94 Proceedings*, Kyoto, Japan.

Silberztein Max. 1996. *Levée d'ambiguïtés avec NooJ*, in BULAG #21. Université de Franche Comté.

Silberztein Max. 1996. *Analyse automatique de corpus avec NooJ*, 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 NooJ, 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 NooJ, 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.

NooJ

Check out NooJ's WEB site regularly at www.nooj4nlp.net.

The main page provides links to NooJ related events and conferences; some of them have published their proceedings on line.

Follow the link "Doc and Help" to download the manual and electronic versions of several papers on NooJ, reference and links to articles on NooJ.

Follow the link "Community" to download several tutorials and a documentation in French.

Follow the link "References" to display lists of books, papers and articles on NooJ.

The NooJ Theoretical and Methodological Background

Silberztein M. 2015. Silberztein, M. (2014). *Formaliser les langues : l'approche de NooJ*. ISTE Ed.: Paris. (426 p.).

Proceedings of NooJ Conferences

Mario Monteleone, Maria Pia Di Buono, Johanna Monti, Max Silberztein Eds. *Formalising Natural Languages with NooJ 2014*. (22 articles). Cambridge Scholars Publishing: Cambridge, 2015.

Svetla Koeva, Slim Mesfar, Max Silberztein Eds. *Formalising Natural Languages with NooJ: selected papers from the NooJ 2013 International Conference.* (18 articles). Cambridge Scholars Publishing: Cambridge, 2014.

Anaïd Donabédian, Max Silberztein Eds. *Formalising Natural Languages with NooJ: selected papers from the NooJ 2012 International Conference.* (31 articles). Cambridge Scholars Publishing: Cambridge, 2013.

Vučković Kristina, Bekavac Božo, Silberztein Max Eds. 2012. Selected Papers from the 2011 International NooJ Conference. Cambridge Scholars Publishing: Newcastle (22 selected articles, 260 pages).

Gavriilidou Zoe, Chatzipapa Elina, Papadopoulou Lena, Silberztein Max Eds. 2011. Selected papers from the NooJ 2010 International Conference and Workshop. Univ. of Thrace Ed, Greece (21 selected articles, 279 pages).

Ben Hamadou Abdelmajid, Mesfar Slim, Silberztein Max Eds., 2010. *Proceedings of the 2009 International NooJ Conference*. Centre de Publication Universitaire, Univ. de Sfax (23 selected papers, 328 pages).

Kuti Judit, Silberztein Max, Varadi Tams Eds., 2010. *Proceedings of the 2008 International NooJ conference*. Cambridge Scholars Publishing (18 selected papers, 296 pages).

Blanco Xavier, Silberztein Max Eds, 2008. *Proceedings of the 2007 International NooJ conference*. Cambridge Scholars Publishing (18 selected papers, 296 pages).

Koeva Svetla, Maurel Denis, Silberztein Max Eds. 2007. *INTEX et NooJ pour la Linguistique et le Traitement Automatique des Langues*. Proceedings of the 4th (Sofia, 2003) and the 5th (Tours, 2004) NooJ conferences: Presses Universitaires de Franche-Comté, (24 selected papers, 481 pages).

Various papers (not in the proceedings above)

Silberztein Max, 2004. NooJ: A Cooperative, Object-Oriented Architecture for NLP. In *INTEX pour la Linguistique et le traitement automatique des langues*. Cahiers de la MSH Ledoux, Presses Universitaires de Franche-Comté.

Mesfar Slim, Silberztein Max, 2008. Transducer minimization and information compression for NooJ dictionaries. In Proceedings of the FSMNLP 2008 conference, Frontiers in Artificial Intelligence and Applications. Ed IOS Press, Netherlands.

Silberztein Max, 2007. An Alternative Approach to Tagging. Invited Paper In *Proceedings of NLDB 2007*. LNCS series, Springer-Verlag Eds, pp. 1-11.

Silberztein Max, 2005. NooJ's Dictionaries. In the Proceedings of LTC 2005, Poznan University.