Distributing and Porting General Linguistic Tools

Damien Genthial, Jacques Courtin and Jacques Menézo
TRILAN-CLIPS, IMAG-Campus, BP 53
F-38040 GRENOBLE Cedex, FRANCE
{Damien.Genthial Jacques.Courtin Jacques.Menezo}@imag.fr

Abstract

Our main motivation is to build general and adaptable linguistic tools and we have faced the problem of their portability. We first make a quick description of the linguistic tools we have at hand and we explain why linguistic tools, unlike other software tools, present particular portability problems. We then discuss code portability and also data portability and we describe the method we have used for a French lexicon, showing that portability leads to a more "natural" computational lexicon. We then propose the use of a command language to interface the tools with more complex applications and we show that this technique facilitates integration of tools from various sources, entails a better exploitation of linguistic resources and makes easier the distribution of tools on several machines.

1. Introduction

Our main motivation is to build general and adaptable linguistic tools and we have faced the problem of portability of these tools. The problem has raised sharply when we decided to implement a distributed version of the tools. The idea is to have bricks to build complex linguistic systems and to make possible, and easy, communication between bricks. We have three points in mind:

- integration of tools from various sources: the linguistic system must not know the details of the internal architecture of the tools it uses, so it should be easier to substitute one tool by another (for example you can easily change the morphological parser);
- better exploitation of linguistic resources by embedding them in very general tools;
- possibility of a distribution on several machines of a net, allowing tools to be shared by several users (and thus the cost can also be shared).

After a quick description of the tools we have at hand, we will explain why linguistic tools, unlike other software tools, present particular portability problems. We will then discuss the user interface portability and we will propose a simple method which, making this portability easier, is also a good way towards distributed tools and easy communication between them.

2. Linguistic tools at hand and motivation

We have a complete morphological system based on a general finite state transducer. Its main characteristics are its reversibility (the same data are used for parsing and generating) and its adaptability (the system includes editors which permit easy and interactive modification of the data). This system is operational on PC and Macintosh architecture with a real size French lexicon, but switching from one architecture to the other is a painful process, mainly because switching the system also implies switching the lexicon (see next section).

We also have three lexical correctors: one based on similarity keys, another on phonetics and a third, more original, which correct flexional errors in French. All these tools are operational on PC architecture only.

Finally, we have two syntactic parsers which build dependency structures. One is based on the notion of dependency relations and is very fast but has a limited power of expression. The other uses typed-feature structures to increase this power but pay the bill with slower parses. Both works on PC and Macintosh.

The interesting point comes when we decided to make all these tools available on Unix systems. The goal is to gain flexibility and power by a distribution of the linguistic tools in a client/server architecture. With such an architecture, tools are more easy to use and are sharable among applications. For example, as proposed by (Genthial, 1994) a phonetic/graphic transducer which implements a lexical correction, can also be used in a syntactic corrector to determine the most probable correction. Tools can also be dispatched on different machines, such that one can, for example, write on his PC or Mac and use the linguistic tools of a Unix server.

So the problem to solve looks like a software engineering one: we have a lot of code,

written in different programming languages on two different machines, and we want to implement it on a new architecture. But we have add a more heavy constraint: we want that code and data obtained on the new architecture (Unix) can easily — by easily we mean in only a few minutes — be put back on the other ones (Mac and PC).

3. Code and data portability

Code portability is not specific to computational linguistics, it is a well known problem in the software engineering domain, but implementing a linguistic application means also implementing an important amount of data and thus raise the problem of data portability. Considering morphological level for example, implies coding a lexicon including words with their category, their morphological properties,... Categories and properties are symbols chosen by the linguist and he can always choose symbols which can be expressed in the same way on different machines, and thus be portable. But words are character strings, coded with the character set of the machine used and so the portability of the word list rely upon the portability of this character set. The ASCII character set, which is the basic set on almost every machine, is fully portable but it does not contain every character of every natural language: using the French *é* or *à* or *ç* implies the use of an extended character set which is not portable.

After a small discussion on code portability, we will present a method to achieve data portability.

3.1. Code

Code portability is heavily tied with the programming language used for writing programs: the more portable is the language, the more portable is the code. That is the reason why we had chosen the Pascal programming language in the early 70's : the language was well defined and we used only the standard features. But the language has evolved and the evolution leads to incompatibility between versions.

On the contrary, the C language has been standardised in 1989 by the ANSI and we can now speak of a real portability of code from one architecture to another. We have then chosen to use C and the biggest part of rewriting Pascal units to C modules has been achieved by a Pascal to C translator.

But one problem remains: we want to put back the C translation on the original machine with minimal work, and the original code includes a user interface with pull-down menus and dialogues which are impossible to translate as is. So we have made an effort to cut the C version in two parts:

- the user interface, which is heavily unportable and must be rewritten on every machine (see section 4 for a discussion on interface portability);
- the tool kernels, written in strict ANSI-C.

Thanks to the language standard, the kernels (about 8000 lines of code) have been compiled, without changing even a comma, on Macintosh, PC and two different Unix machines.

3.2. Data

Two kinds of data may be used in linguistic applications: textual data and binary data. Most of them are textual because they can easily be printed, displayed and modified with the standard tools of the host system. But sometimes you need to compile data to gain efficiency: the application becomes faster and use less disk space.

Binary data in linguistic applications are for example integers, bit vectors coding properties, floating-point numbers coding statistics and so on. Their portability is not a real problem because one can easily translate them in textual form on the original machine, put this form on the target machine, and compile them back.

As said before, portability of textual data rely upon portability of the character set, so using ASCII set ensures a great portability but forbids writing special characters. Such special characters are all French accented letters (*à, â, é, è, ê,...*) which can be coded (and typed) on every machine but the codes are different from one machine to another. Moreover, all special character codes are above the ASCII maximal code and this entails a disturbing side effect: when sorting words of a lexicon you get all words starting with an accented letter at the end of the list (see example on Figure 1).

errer
oui
outre
ouïe
vent
érudit
ôter

Figure 1 : Sorted accented strings on a PC

When the lexicon is big enough, the word *érudit* is far from *errer*, which is computationally sounded but unacceptable for the common user.

We have then defined an internal code for special characters based on the ASCII character set. The code is a reduced version of one defined by GETA in (Boitet, 1982) an accented letter is coded with the letter without

accent, a vertical bar, and a number corresponding to the accent (see examples on Figure 2)[1].

à → a|2
â → a|3
é → e|1
è → e|2
ê → e|3
...

Figure 2 : Examples of the code for accented characters

All textual data are then completely portable provided that source and target machines use ASCII. But there are two drawbacks: you can not ask the user to learn this code and you can not use the standard string comparing functions. For the first problem, we simply write two procedures: one for reading strings and one for writing. Their purpose is to translate from one representation to the other such that the user has no need to know the internal code: he can type special letters as usual on his keyboard. For the second, the solution is to write our own comparing function, which is not so difficult and have an advantage: we can implement a "natural" order on words (the order used in paper dictionaries). We then obtain a human sounded order which can also have a computational advantage in correction systems. Consider for example the four French words *cote*, *côte*, *coté* and *côté*: their proximity in the lexicon is a guarantee for a corrector to find the correction if one is used for the other, guarantee that you cannot have with the preceding order (765 root words between the roots *cote* and côté in our French root dictionary, which contains a total of 35 000 roots).

With this code, we get textual portability of data and a natural dictionary order which is preserved on all machines where the dictionary is implemented.

4. Driving tools with a command language

Once you have achieved the portability of your software kernels, you are faced the portability of the user interfaces. Here you have two choices:

1. write portable interfaces by using very simple textual interactions with the user so that you can write the code in ANSI-C;
2. write a modern interface, heavily tied with the graphical interface of the host machine, and partially or completely rewrite it each time you want to implement it on a new architecture.

We have chosen to proceed in two steps:

- first make the first choice even if we get a very poor user interface, not acceptable on modern graphic computers; such interfaces are very easy to write and permit at least to debug the tools.
- then make the second choice, try to minimise the rewriting cost and, moreover, to make the kernels completely independent of the interface.

To minimise the rewriting cost, we use a graphical library which is freely available and portable from one machine to another.

To make the kernels completely independent of the interface, we propose to have a user interface which is strictly limited to communications with the user. The architecture is a client/server one, where the user interface (the client) calls the kernels (the servers) for linguistic treatments (see Figure 3).

CLI: Command Language Interface

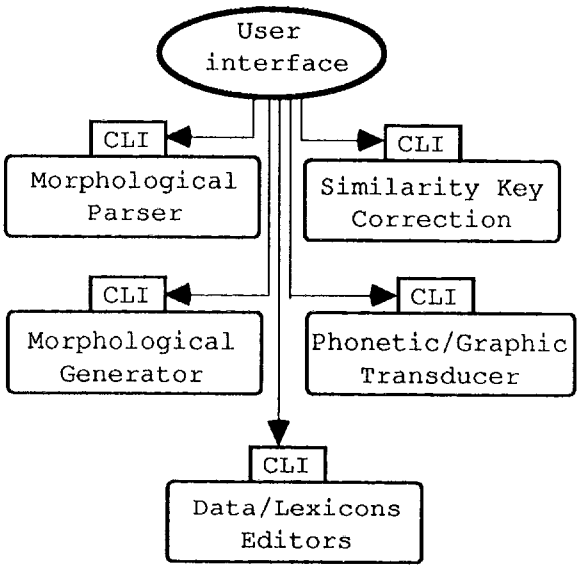

Figure 3 : Distributed Architecture with separated user interface

You can imagine as much clients as you need, for example:

- one for a lemmatiser which calls only the morphological parser and generator;
- one, more complex, for a detection/correction system, which uses all tools to produce correction of lexical errors;

[1] The code defined by the Text Encoding Initiative (Sperberg, 1994), derived from SGML, is usable for electronical transfer, but a little cumbersome for a lexicon which might contains as much as 200 or 300 thousands words

- one, with pull-down menus and windows, devoted only to the editors (modification of the lexicons, of the linguistic data,...).

Of course, all interfaces are sharing the tools with the others and it must be easy to add a new tool to an interface (for example a new correction method) or to substitute a given tool by an other (one can change the phonetic/graphic transducer to get an improved version).

To obtain this flexibility and to make possible the distribution of tools (on the same machines or on all a net), we propose, as (Boitet, 1994) in the white-board architecture, to add a manager on each module. Our manager take the form of a textual command language which is used to drive the module (Antworth, 1990) has used such a command language interface in PC-KIMMO.

The general form of a command would be the following:

```
verb(arg1 => param1;
     arg2 => param2;
     ...)
```

where `verb` is the command and where arg_i and $param_i$ are respectively the names and the values of its parameters.

Parameter values could be integers, floating point numbers, booleans, objects (denoted with the same syntax as a command), or a list of the preceding.

Examples:

```
Parse(string => "to_be_parsed")

Generate(
  word => "aimer";
  filter => filter(
    category => "verb";
    variables => ["present",
                  "singular",
                  "3rd_person"])

List(dictionnary => "dict_name")

Add_dictionnary(word => "to_add";
                like => "paradigm")
```

Each tool must be build on the same frame: it reads only from one input stream (its standard input) and write to only one output stream (its standard output) and the main algorithm is an interpreter.

Using such a command language interface entails 4 main advantages:

- it can be used as the only (but rough) interface for a given tool;
- you can write programs in this language and thus automate the use of the tool;
- the interpreter does not use machine specific feature so the entire tool can be written in strict ANSI-C and thus be heavily portable (without changing a comma);
- connecting the tool to a more sophisticated interface program is very easy: it requires only the ability of passing text from one application to the other. You can for example put a morphological parser on a machine such that it can be called by electronic mail: you send the string to be parsed in a mail and the answer contains the words, with their category and properties.

5. Conclusion

We have used the portability frame presented in this paper for the main tools of our system: a morphological parser and a morphological generator, which use a root and endings lexicon to parse or generate about 250 000 French forms. The lexicon must be uncompiled and compiled back when porting from Mac to PC but the whole process does not take more than a dozen minutes. On the contrary, thanks to the similarity in their architectures, the same lexicon can be used on Mac and on Unix machines.

Concerning the code, we have now portable versions of the tools mentioned above, plus a lexical desambiguer and a lexical corrector using similarity keys. We are able to deliver libraries for these tools (and their data for French) on Mac, PC and Unix.

References

E.L. Antworth (1990). PC-KIMMO : A Two-level Processor for Morphological Analysis, *Summer Institute of Linguistics*, Dallas, Texas.

Christian Boitet (1982). Le point sur ARIANE-78. *Rapport ADI 81/423*, GETA-Champollion et CAP SOGETI France, Grenoble.

Christian Boitet and Marc Seligman (1994). The "white-board" architecture: a way to integrate heterogeneous components of NLP systems. *CoLing'94, Kyoto, Japan, August 94*, Vol. 1, pp 426-430.

Damien Genthial and Jacques Courtin (1994). Towards a More User-Friendly Correction. *CoLing'94, Kyoto, Japan, August 94*, pp 1083-1088.

C.M. Sperberg-McQueen and L. Burnard (1994). Guidelines for Electronic Text Encoding and Interchange. *in press*, Chicago and Oxford.