UNIVERSITY OF MURCIA

International Journal of English Studies

IJES

www.um.es/ijes

Developing Software for Corpus Research

*Oliver Mason**
University of Birmingham

ABSTRACT

Despite the central role of the computer in corpus research, programming is generally not seen as a core skill within corpus linguistics. As a consequence, limitations in software for text and corpus analysis slow down the progress of research while analysts often have to rely on third party software or even manual data analysis if no suitable software is available.

Apart from software itself, data formats are also of great importance for text processing. But again, many practitioners are not very aware of the options available to them, and thus idiosyncratic text formats often make sharing of resources difficult if not impossible.

This article discusses some issues relating to both data and processing which should aid researchers to become more aware of the choices available to them when it comes to using computers in linguistic research. It also describes an easy way towards automating some common text processing tasks that can easily be acquired without knowledge of actual computer programming.

KEYWORDS: Corpus linguistics, corpus analysis, developing software, programming

**Address for correspondence:* Oliver Mason, University of Birmingham, UK. Tel: 0121 414 6206. E-mail: O.Mason@bham.ac.uk

I. INTRODUCTION

Over the past twenty or so years the use of the computer in linguistics has changed from simply being a glorified typewriter for writing up articles to an essential research tool. The increase in processing power and storage space has also made it possible for researchers to access electronic corpora on an unprecedented scale, first on networked servers, but increasingly on individual personal computers. At the same time, however, the set of skills taught in Universities has not changed enough for linguists to keep up with such developments.

In this article I will discuss issues of using computer software in corpus analysis, both readily available programs and custom-built ones. This also includes data formats, as there are two components to data processing: the software for the processing, and the data to be processed. Both components need to be compatible for the process to work.

II. OFF THE SHELF OR WRITING YOUR OWN?

The first question one faces when researching authentic language (or any other area which requires looking at larger amounts of data in a particular way) is what software to use. We assume that a 'manual' analysis, that is, without using the help of a computer, is out of the question due to the size of the material to look at. The choice of software obviously depends on the nature of the analysis, and the availability of appropriate programs. Performing a 'tried and tested' kind of analysis, where well-established procedures are applied to new corpora would be a candidate for using existing software. Most common procedures (such as concordancing or computing collocations) have been implemented as ready-to-use programs.

Using readily available software, however, might not always be the best solution. As with most analytical procedures there are parameters, decisions that need to be made which influence the outcome to a potentially large degree. With a basic procedure such as concordancing this is not much of an issue, as there are few decisions involved: do you fold upper and lower case? What about inflected forms of the word you are looking at? It is usually trivial to run the analysis several times including potential variant forms if they are not included, though it might be harder (if not impossible) to filter the results if variants are included and this feature cannot be switched off. At present it would also be very hard to look at the use of horticultural terms in journalism if one cannot distinguish between upper and lower case for words such as *bush* and *brown*.

More complex analyses involve more decisions. Collocations, for example, require a whole host of parameter settings (see Mason, 2000a), but often software does not allow the user to change these values, or does not even make them public (e.g. as part of the documentation). This lack of control not only limits the kind of analysis one can perform, it also makes it impossible to replicate studies reliably. In the field of mathematics this

dilemma has lead to the creation of open source software ('Sage', see URL below) for mathematical computing, as commercial applications could not be scrutinised to the degree it would be necessary.

So, there are even issues with available/existing software, but it at least allows you to perform a number of tasks that would otherwise be impossible. But as soon as one departs from the common path, things begin to look even worse. Corpus linguistics is still a comparatively young discipline, and its methodology is still in flux; new procedures are developed, old ones are modified or extended, new challenges come up with an expanding range of languages being analysed, and so forth. As a consequence, much research has to make do without software being readily available for a particular purpose. This also applies to any innovative research that explores new ways of looking at language data.

As soon as the established procedures are found sufficient, one has to use custom software that implements things in a different way, or that adds features that were not available before. There are several options for doing so, and they involve different amounts of effort to be realised: First, there could be an option for corpus software to be extensible via plug-ins. Here we would not need to create a completely new program, but would simply implement a new procedure as a module, which can then be added to the existing software. Unfortunately this requires more complexity on the part of the original program, and not many programs are thus extensible in such a way. It could also be problematic if the new procedure requires fundamental changes throughout the core of the software, for example with the way a 'word' is defined (is *don't* one word or two?).

On the opposite extreme we could write a new program from scratch in a common programming language (see below for a discussion of different kinds of modern programming languages). This gives us complete freedom in designing our application, but also requires most effort, and is most time-consuming. There is a lot of literature on software engineering, but despite years of research in the area, most commercial software projects run over time and over budget. Eckel (2007) estimates that 50-80% of all programming projects fail. This even happens to open source ventures; for a discussion of the difficulties of writing software that does not require a PhD in computer science to understand see Rosenberg (2007).

Another option would be to start with a set of modules which can be combined like building blocks to create a custom program. This is the original promise of object oriented programming, where modularity in the design allows re-use of components, and easy combination of such components into new systems. Here we need more effort than in the first case, but we also have more flexibility. And if a module doesn't do what we want, we can just replace it. However, this dream of building software out of modules like buildings out of blocks of lego has not come true yet, mainly because so much depends on design decisions which will be different for different projects, and there is a lot of variability in the way real-world data can be represented within a computer's memory, complicating the procedures that then have to handle that data.

In programming there is a trade-off involved between the level of flexibility and the work required to achieve a goal (see Mason, 2000b). A simple program is limited in its capabilities, but usually easier to use than a sophisticated one which might allow the user to perform a broader range of tasks. This trade-off also applies on 'the other side': Adding flexibility to a program requires more effort than restricting the possible options. Especially for exploratory research it is often not possible to spend a lot of resources (either time or money) on developing completely new software, when it turns out that a number of changes and adaptations are required as the project develops. But it is also unsatisfactory to rely on existing software that limits the possibilities of getting a fresh view of the data.

As Scott (this volume) shows, the driving force behind software development in linguistic research often is the need of individuals who are able to do such work in their spare time. It is very difficult to get funding for software development in an environment dominated by an arts/humanities mindset that research is something done sitting either in the library or at a desk, using a computer only as a glorified typewriter to simply write a research paper. Most software available has thus been created by 'amateur enthusiasts' (e.g. WordSmith), research programmes in computer science that involve text processing (e.g. IMS Corpus Workbench), or in rare cases are commercial developments accessible to researchers (e.g. Cobuild's 'lookup').

Corpus linguistics generally suffers from a lack of programming skills, as most practitioners originate from linguistics rather than from disciplines such as computer science where undergraduates receive training in programming languages. As a consequence, developing software for exploring new methodology becomes a non-trivial task, limited to those few corpus linguists who are either self-taught programmers or who originate from other disciplines (e.g. computational linguistics). This can seriously stifle innovation, as most research relies on existing implementations of well-known procedures, often with severe limitations.

Another consequence of the general lack of programming skills is that much research has do be done largely manually. Concordance lines will have to be individually inspected by the human researcher, which can be a very time-consuming task, and such analysis does not scale, i.e. it is limited with respect to the amounts of data that can be analysed.

Furthermore, lack of technical awareness has the side-effect that researchers often do not know *what* the capabilities of linguistic software are nowadays. Despite many problems with getting a grip on natural language processing there is a range of tasks that can be solved with a comparatively high rate of success. Hunston and Francis (2000) discuss a range of syntactic phenomena they state would be outside the scope of computational analysis due to some ambiguity in the surface structure, when it is in fact fairly easy to get a computer to handle the examples they give. But if it is perceived that a computer program will not be able to solve the problem, then one will not even attempt to find a solution and despite the possibility of computational assistance the researcher will have to fall back on manual analysis.

In phonetics a different set of problems is faced; there are a number of programs available for acoustic analysis and digital signal processing, but the algorithms used are generally too complex to be implemented by those who are only interested in the application of the procedures (e.g. calculation of the fundamental frequency of an utterance). In this case there is not really any choice, and one has to accept the situation just like a biologist who cannot build their own microscope.

To summarise, programming skills are important in today's corpus research when one attempts to go beyond well-established procedures. However, not every practitioner has to push forwards the boundaries of the discipline, nonetheless, an awareness of the limitations of currently available software is necessary for planning a research project. Furthermore, knowing which tasks can be performed by a computer and which tasks require human involvement is also useful. Being able to implement analytic procedures in form of a computer program cannot really be expected from every corpus linguist, but that means that these skills need to be imported from other disciplines.

III. DATA FORMATS

Few corpus linguists will actually have to do their own programming, but many will gather their own corpus data for analysis. For that reason, knowledge of appropriate data formats is more important and relevant, and should be part of any syllabus related to empirical language study.

Obviously, software and data formats are closely interlinked, as it is the computer programs that read in data in order to process it. If the data is not in a format that the program can interpret correctly, then it will not be able to work with it. This is a general problem in software engineering, and over the years a number of standards have evolved which ensure that data can be stored in ways that are independent of a particular piece of software.

The main benefit of having standardised formats is that one is not limited to a specific program for accessing the data. If a program has an idiosyncratic data format, the data cannot be used with any other program without it having to be converted into a different format; this is often a tedious and error-prone procedure, which can lose information if the expressive power of the formats is different.

In the early days of computing, capabilities were limited, and so was storage space. Computers usually had a limited character set only, and text was stored using just upper case letters. If the difference between upper case and lower case was relevant, additional codes had to be introduced. In the Brown corpus, for example, capital letters were prefixed with an asterisk (Francis and Kucera, 1979). Other corpora (e.g. Sinclair et al, 2004) did not distinguish between upper and lower case, and as a consequence that information is now

lost. This, however, is not a problem with spoken corpora where the distinction does not exist in the first place.

Text corpora are not usually formatted in the same way as word processing documents, so marking changes in the typeface (bold, italics, etc.) are rare. But often structural information is relevant, such as paragraph boundaries, headlines, and text boundaries. Added to that there is meta-information, such as author, text source, date of publication, copyright information, and so on. This data needs to be represented in a way that allows the researcher to easily keep it separate from the actual texts.

Research in text processing and publishing has produced a comprehensive format for this purpose: SGML (Standard Generalized Mark-up Language, Goldfarb, 1991). With storage space becoming cheaper and more available, mark-up can afford to be more verbose and explicit; instead of cryptic (and more restrictive) one-letter-codes it is possible with SGML to label information in a way that even 20 years later can still be understood. This is done using *tags*, which have different forms depending on whether they mark the beginning or end of what they enclose. The information for a particular text could then look as follows:

<document>
<author>Carolus Magnus</author>
<title>Introduction to Lower Case</title>
<translator>N.N.</translator>
<text> … </text>
</document>

Shorter labels as used in pre-SGML times can often be ambiguous: does ‘A’ stand for *author*, or perhaps *affiliation*? Using <author> removes any possibility for misunderstandings. Furthermore, SGML defines a complete format, so no other implicit conventions need to be introduced in order to separate data from content. And being a standard means that there are many different programs and modules available to read in SGML data in any programming language, ready to include in newly-written applications.

SGML, however, still is a rather complex format, allowing for short-cuts to omit tags. Given a formal grammar of the document, one could for example leave out the closing </author> tag: the following tag indicates a title, and a title cannot be part of the author element, so implicitly the author element is terminated. This was necessary at a time when storage space was still more limited as it is nowadays, and the amount of just the closing </w> tags saved for instance in the first version of the BNC (four characters per token, altogether about 400 megabytes) was bigger than PC hard drives common at the time.

A simplified variant of SGML is XML (extensible mark-up language); this requires explicit tags, and is thus more verbose, but in consequence it is a lot easier to process. XML has developed into something of an industry standard for storing data, applied not only to

corpora. There are a number of different modules available for processing XML data in many programming languages.

One often overlooked advantage of XML is that both computers and humans can make sense of it. It is also not bound to a particular computer, operating system, or application. Storing corpus data in a word processor on the other hand severely limits interoperability, as typically the exact version of the word processing software is required to view the data. Such problems are easily avoided by using XML.

In the past corpora have sometimes been stored in a compressed format for efficiency, e.g. in the Bank of English corpus each token was represented by a number. This made searching very efficient, as effectively each token now had the same length, and all that was required for finding a word was a translation table from the token to its index number, and a list of all the places where the token occurred in the corpus. But the downside is that the data itself was no longer in text format, and changing the text, or adding new data to the corpus was much more complex. Here we have another trade-off, ease of access *vs* processing efficiency. Luckily, processing XML data has become more efficient in recent years, a benefit of it having become a widespread standard.

When collecting corpus data, researchers should make use of all the tools available for handling XML. There are even specifications available from the Text Encoding Initiative about encoding corpora in XML (see Corpus Encoding Standard URL). Encoding data in this format means that all corpus tools that are capable of working with this format can then be used to process the texts.

IV. PROGRAMMING LANGUAGES

While it is not strictly necessary to be able to program a computer when doing text analysis, it is still a very useful skill that can save a lot of time when doing repetitive tasks that one would otherwise have to do by hand. Writing computer programs is not hard; there are a wide variety of programming languages available, some of which are well-suited to beginners and do not have a steep learning curve. All these different languages have both advantages and disadvantages, and some languages are better for certain applications than others. There are several general-purpose languages, some that are mainly useful for mathematical computations, and others that work well for text processing. This section will briefly discuss different types of languages, focusing on those which would be suitable for a beginner in the area of text processing.

IV.1. Interpreted or compiled

At the heart of the computer is a microprocessor (CPU) which executes very low-level instructions at a very high speed. These instructions are very basic: mainly moving numbers

around in memory, adding and subtracting values, and so on; thus they are rather far removed from what one would want the computer to do. A simple task (eg comparing two words for equivalence) needs to be broken down into a fairly large number of these machine instructions. In order to make programming easier there are more powerful instruction sets at a much higher level, more suitable for humans, which are translated into those lower level commands. In principle there are two different approaches to doing this: either at the time when a program is written, or when it is executed. Programming languages which are translated before the program is executed are *compiled*, and the program which does the translation is called a *compiler*. Those which are translated when they are executed are *interpreted* languages, and they require a program called an *interpreter* to be run. Some languages can work in both of these modes.

IV.2. Compiled Languages

Compiled languages are generally faster, as the translation happens before the program is actually executed. Errors in a program are often caught by the compiler, which reduces the type of bugs that a program will have. On the downside the development process takes longer, as the program has to be re-compiled every time something is changed. The compiled program is also usually tied to a particular platform/operating system, and would need to be re-compiled when it needs to be run somewhere else. Examples for compiled languages are Fortran, Pascal, and C.

IV.3. Interpreted languages

Interpreted languages are translated as they are needed. That makes development very fast, as it is easy to change parts of the program code and run it again. No compilation is required, so there is no delay between saving the change to the program and running it. On the other hand, there are errors that a compiler would catch, which can slip through, e.g. in a segment of the program that is not executed very often. Interpreted languages are flexible and allow for portable code, as interpreters for the common languages are available for many platforms. Examples for interpreted languages are Python, AWK, and Perl.

Java is a special case, as are some variants of Pascal: here the program code is compiled into an intermediate ‘bytecode’, which is halfway between the original source code and the machine executable code. A run-time system then interprets the bytecode, which can be run on many platforms. Here the interpretation process is faster than with fully interpreted languages, as it does not require a lexical or syntactical analysis—this has all been done during the compilation stage. Furthermore, the run-time environment can perform optimisations specific to the actual computer that it runs on, so that Java programs can even be faster than fully compiled ones.

A variety of other programming languages make use of the execution model/platform of Java: they have been designed to be compiled into Java bytecodes, so that they share the platform independence while offering different features; at present there are more than 200 systems available (see URL below), which include many popular programming languages. One of these languages is Scala, which exists both as a compiled and an interpreted variant, and combines features of Java with aspects of functional programming.

V. BOTTLENECKS

A computer consists of various components, the most important ones being the CPU (central processing unit), the (volatile) memory, and the hard-disk. The components communicate through a so-called bus, which shuffles data from one to the other. Understanding the interaction between those components can be very important.

The CPU is the fastest, memory is slower, and the hard-disk is the slowest of the three. Other I/O (input/output) devices are even slower, e.g. CD-drives. This has consequences for text processing, as one typically deals with a lot of data, which needs to be stored somewhere.

Accessing corpus data from a CD (or a DVD) is not a good idea, as the CPU has to spend a lot of time idle, waiting for the data to arrive from the drive. Storing corpora on a hard disk is thus a better idea, as the CPU, which is where the computational work gets done, does not have to wait as long due to the higher drive speed. Ideally the data should be stored in memory, but capacity is often a problem. One option is to read chunks of text into memory which are then processed, which might speed things up; but generally the I/O bottleneck is the biggest issue with large-scale text processing.

A more recent development is that CPU speeds seem to reach the end of the growth limit. In the past, CPUs have become faster and faster, from 4.8 MHz of the original IBM PC (introduced in 1981) to around 2.4 GHz of current PCs—a five-hundred-fold increase. But physical constraints (eg heat and space) mean that CPU speeds will not continue to improve in the same way in future. Instead, CPUs now have multiple *cores*, i.e. there is more than one actually processing unit. Certain tasks can therefore be executed in parallel, yielding an increase in speed proportional to the number of cores available. In order to exploit the potential for parallel programming, the software needs to be capable to be parallelised, which adds another dimension to the set of problems a developer has to struggle with. It might then even be the case that a task will have to be approached in a way that is *slower* than the traditional one, but *faster* because some parts can be executed in parallel. However, this is unlikely to be of concern for the average corpus linguist.

There are other ways of making sure the I/O bottleneck does not impact too much on a program's performance. For example, a concordancing program could search a corpus sequentially for all occurrences of the requested word. This means the full text has to be read

in and matched. It is much faster to create an index of all words, which can then be used to locate the occurrences before even beginning to read the text. If a word does not occur, the result will be instant, whereas a sequential search would still have to look through the whole corpus. Creating an index takes time, but it is worth it due to the time savings afterwards. Generally the indexing time will be slightly more than one sequential run through the data, as the full text has to be read to find the positions of all words. Space requirements are more important, but modern compression techniques (see Witten, Moffat, Bell, 1994) can keep this to a minimum.

When writing software for corpus analysis one should take care to keep in mind the points where processing will be slowed down; this will mainly be where data has to be read from a disk, or over a network connection. Sometimes rearranging the order of processing steps can make a big difference if repeated disk accesses can be avoided. For the non-programmer this will often not be too much of an issue, but it is worth, however, to spend some thought on arranging the data in the right way, such as copying corpus data from a CD to the hard disk.

VI. COMPUTING WITHOUT PROGRAMMING

Learning a programming language is a hard task, as one not only needs to get to know the syntax and the supporting libraries, but one also has to adapt to the 'mindset' of the computer: unlike human beings the parser which analyses the source code is unforgiving and will complain about missing brackets, and a misplaced semicolon might alter the result in unforeseeable ways. The computer cannot second-guess the programmer's intentions and thus takes the written commands absolutely literally. As a consequence, programs are often full of mistakes, and ironing them out or even understanding why the program does not behave in the intended way can be very time-consuming and frustrating. This can be seen as the price of power: one can tell the computer what to do down to the last detail, but it involves a lot of effort to get to the stage where one can get the computer do actually do what it should be doing.

However, there is another alternative which does not involve knowledge of an actual programming language, but merely mastery of a number of individual tools. On every Unix system there are a number of useful small programs, which can easily be combined to perform a lot of common tasks in corpus analysis. These text tools can also be used on a DOS command line, but they are at their most powerful in the Unix environment. This would require a computer that for example runs a version of the freely available Linux operating system (which can be installed on any PC), or one of the recent Apple computers running Mac OSX, which is based on a variant of Unix.

The text tools work on a command-line, so one would need to open a 'terminal' window to get one. Each of those tools reads input from a channel, performs an operation on the

data, and then produces output to another channel. The output of one tool can be fed into the input of the next tool, creating a so-called 'pipe'. The most useful tools for text processing are:

- cat to read a file
- tr to translate characters
- sed to do search/replace
- sort for sorting
- uniq for removing duplicate lines
- grep for pattern matching
- wc for counting words, lines, and characters

For more complex tasks it is possible to create small scripts using tools such as awk (for a brief introduction see Barnbrook, 1996) or perl (Wall and Schwartz, 1991) which can be kept very simple and thus easy to write. We will now look at some examples for simple processing tasks. The tools are joined up using the vertical bar |, and the greater-than symbol > redirects output into a file. The input files in each case is a plain text file.

VI.1. Creating a frequency list from a text

```
cat mytext | tr -C "[A-Za-z0-9]" "\012" | grep -v "^$" | sort
| uniq -c | sort -nr > list
```

This single line takes a file called "mytext", pipes it into the tr tool. This replaces all characters which are neither letters or digits by line-breaks. The output will contain at most one token per line, but also empty lines. These are removed by grep, which filters these out: the caret is a special character that stands for the beginning of a line, whereas the dollar sign signifies the end of the line. The expression "^$" thus matches all lines where the end follows the beginning, i.e. those that are empty. The -v option causes matching lines to be suppressed from the output, so that all empty lines are then removed. The next step is to sort the tokens (alphabetically). Then we remove duplicates, but with the -c option we instruct uniq to count how often each token occurs, and the number will be put before the token. Then we sort again, but this time in reverse numerical order, so that the most frequent tokens are at the top. The output is then placed in a file called "list".

If we have the single line "the cat sat on the mat." in our file we get the following output:

```
2 the
1 sat
```


1 on
1 mat
1 cat

If we want to fold upper and lower case we could insert the command tr "[A-Z]" "[a-z]", which translates all upper case characters to lower case. Then the input “The cat…” (note the capitalised “The”) would lead to the same output as our example.

At first this might almost look like black art, using a complex command line to turn a text into a sorted frequency list, but it can easily be decomposed into individual steps. The output at each stage can easily be inspected, so one can see how the various tools interact to create the final result.

VI.2. Counting the number of words ending in -ing

```
cat mytext | tr -C "[A-Za-z0-9]" "\012" | grep "ing$" | wc -l
```

The first part of this command line is identical to the previous example. Only this time the grep command filters out those words where “ing” occurs at the end of the line. Without the -v option the words *not* matching the pattern are suppressed, so the output here would be all tokens where the pattern applies. We are not interested in the list of tokens, but only in their number, so we pipe this list into the wc tool (which stands for “word count”). The -l option instructs it to count the number of lines in the input, and as we have one token per line it results in their number being printed. If we’re interested in the number of types, we simply put sort -u before the wc: this sorts the input suppressing duplicates (and is thus equivalent to sort | uniq). If we want to find tokens ending in “-ed” we use "ed$" as the pattern for grep; tokens beginning with “un-” would be "^un" (beginning of the line followed by “un”). The grep tool allows one to specify very complex patterns which should be sufficiently powerful for most tasks.

VI.3. Finding words which two texts have in common

```
cat list1 list2 | sort | uniq -c | grep "2 "
```

We assume that the types have already be extracted into a list for each text following the above approach (leaving out the uniq -c from the frequency list commands). The cat command can take more than one argument, so here we combine the two files to a joint output, which is sorted and counted. Word types which occur in both texts will have a frequency of 2, those which only occur in one text but not the other will have a frequency of 1. The grep command thus retrieves those types the two texts have in common.

A similar method can be used to find which words occur in text 1 but not in text 2: we simply add "list1" to the argument list a second time, so that the first part now is cat list1 list1 list2. Types that occur in both texts now have a frequency of 3, those in text 1 only have a frequency of 2, and types that only occur in text 2 have a frequency of 1. We could also extend this if we wanted to compare three or more texts in the same way.

VII. LIMITS OF THE TEXT TOOLS APPROACH

While the text tools are very flexible and can be used to solve a lot of common processing tasks, there are limits. One is that the pattern matching facilities only operate on character sequences, and do not take into account morphological (ir)regularities. The amount of meta-information is also limited, and processing annotations is often difficult, especially if they are stored using XML tags. Matching larger units than single words runs into problems if the elements are separated by line breaks.

There are solutions for these issues, but it means that processing suddenly becomes much more complicated. However, many data processing tasks can usually be simplified so allow the text tools to be used, and then there is always the option of adding awk or perl scripts to the repertoire of tools.

VIII. THE EXPLORATORY CONUNDRUM

In this section we will discuss two fundamental problems with software built for research purposes: how to make sure the software works correctly and how to evaluate the results.

VIII.1. Ensuring Correctness

In software engineering one of the ideas that sound good in theory but do not work in practice is that one can prove the correctness of a program though application of formal logic. Mainly due to the complexity of software this is impractical, and there is an even more fundamental problem: what is the expected behaviour of a program?

The behaviour is commonly expressed in a *specification document*, which is written in natural language. This is then 'translated' into a computer program, and the program works correctly when it follows the specification. But human languages are very imprecise, and programming languages have to be very specific, so there is a natural mismatch between the levels of detail in a specification and the actual software. In some way the software *is* the specification, turning the process of development on its head. The specification should allow a non-programmer to tell a programmer what it is they want; this is an error-prone procedure

every time an expert talks to a non-expert, whether in programming, plumbing, or landscape gardening.

One modern school of software development, Agile development (Subramaniam and Hunt, 2006), realises the difficulties with a specification, and advocates a different approach, user stories and rapid prototyping, an almost constant dialogue with the 'customer' to make sure the development follows the right track and quickly adapts to the inevitable changes in the specification. On a more detailed level, the behaviour of smaller modules of the software is defined through a test, a little routine that feeds the module some input and investigates its output. If the output is as expected, the test passes, otherwise it fails. Initially the modules are implemented as dummies, and thus all tests fail. The developer then implements the routines, and once the tests all pass the program is ready and working correctly.

This approach is called *test-driven-development* (TDD), and it has the added advantage that the suite of tests can be used at later stages to make sure that any changes in the source code do not break the expected behaviour. Adjustments can therefore be made with confidence, as any problems can quickly be identified through failing tests.

However, TDD only works if we know what to expect; it is slightly more difficult to for example create a test for a stochastic tagger or parser, where the outcome depends on so many influences that we cannot predict it with the necessary precision. But apart from that it introduces a good discipline and leads to faster development, as it increases the programmer's confidence that the software still works as expected.

VIII.2. Evaluating exploratory software

Making sure that a program works correctly is hard enough, but it is even harder to evaluate the results when it is not clear what the results should be. Automating an established procedure with well-defined results (such as deterministic parts-of-speech tagging, or parsing, or creating concordance lines) means that the output can be compared to a benchmark; but even in the supposedly easy case of parts-of-speech tagging it is not easy to devise an appropriate metric for determining the success rate: often enough there is no agreement between several human annotators, and no objective benchmark exists. The situation is worse for parsing, where there is practically no overlap between different analyses done manually by different researchers. How can one possibly expect an objective way of measuring a program's performance?

Once we leave the *terra firma* behind and venture into hitherto unexplored areas we are pretty much stuck when it comes to evaluation. Using procedures for exploratory data analysis (e.g. cluster analysis, see Anderberg, 1973) we will always get a result, but we don't know whether it is a good one or not. Some cluster algorithms come with a way of assessing the quality of the result (Kaufmann and Rousseeuw, 1990), but in most cases this is not possible. We have here a situation where we don't know what to expect, and if we

knew what we were looking for the outcome of our search would be prematurely narrowed and we might miss it altogether. This is like fishing in the deep, for some creature you suspect exists, but you do not know what net to use to catch it. With a small one you might catch a lot of smaller fish but not the one you want, and with a big net it might slip through.

A lot of experimental software has this problem, and all we can do is to apply some common sense tests to its output. Look at some boundary cases, check if there are any peculiarities in the patterns. Are there any patterns at all? And if so, do they make sense? Ultimately they ought to reflect either linguistic structures or social realities, so often there is some sense of what constitutes a valid result.

IX. SUMMARY

Research in corpus linguistics depends heavily on computer programming, and unfortunately not many researchers are able to design and develop their own software, instead having to rely on existing programs to do their processing. This not only severely limits the possibilities for exploring corpus data, but also introduces unknown factors into the research, as it is not always obvious how the software will handle certain features of language.

However, programming is also a very complex subject in itself. It is not possible to learn application development as an auxiliary skill in the same way as, for example, academic writing. For that reason there will always remain a reliance on other people to do the development work. On the other hand, it is easily possible to acquire some of the necessary skills to at least create small programs in scripting languages, which will greatly speed up text and data processing.

REFERENCES

Anderberg, M R (1973), *Cluster Analysis for Applications*, New York: Academic Press, Inc.

Barnbrook, G (1996) *Language and computers: A practical introduction to the computer analysis of language*. Edinburgh: EUP.

Eckel, B (2007) *The Mythical 5%*. Address delivered at Neumont University, Salt Lake City. Available on-line at http://www.artima.com/weblogs/viewpost.jsp?thread=221622

Francis, W N and Kucera, H (1979) *Brown corpus manual*, revised and updated version available at http://khnt.hit.uib.no/icame/manuals/brown/INDEX.HTM

Goldfarb, C F (1991) *The SGML handbook*, Oxford: OUP.

Hunston, S and Francis, G (2000) *Pattern grammar: A corpus-driven approach to the lexical grammar of English*, Amsterdam: Benjamins.

Kaufmann, L and Rousseeuw, P J (1990), *Finding Groups in Data*, New York: John Wiley & Sons, Inc.

Mason, O (2000a) "Parameters of collocation: The word in the centre of gravity", in: Kirk, J M (ed) *Corpora galore: Analyses and techniques in describing English*, Amsterdam: Rodopi, p.267—280.

Mason, O (2000b) "A developer's view of corpus linguistics: The CUE system", in: Mair, Ch and Hundt, M (eds) *Corpus Linguistics and Linguistic Theory: Papers from the twentieth international conference on English language research on computerized corpora (ICAME 20)*, Amsterdam: Rodopi, p.233—241.

Mason, O (2000c) *Programming for corpus linguistics: How to do text analysis with Java*. Edinburgh: EUP.

Odersky, M and Spoon, L and Venners, B (2007) *Programming in Scala: A comprehensive step-by-step guide*. To be published in 2008; available as e-book pre-print from http://www.artima.com/shop/forsale.

Rosenberg, S (2007) *Dreaming of Code*, New York: Crown.

Sinclair, J and Jones, S and Daley, R (2004) *English Collocation Studies: The OSTI report* (edited by R Krishnamurthy), London: Continuum.

Subramaniam, V and Hunt, A (2006) *Practices of an Agile Developer*, Raleigh, NC: Pragmatic Bookshelf.

Wall, L and Schwartz, R L (1991) *Programming Perl*. Sebastopol, CA: O'Reilly.

Witten, I and Moffat, A and Bell, T (1994) *Managing Gigabytes: Compressing and Indexing Documents and Images*. New York: Van Nostrand Reinhold.

WEBSITES

Corpus Encoding Standard: http://www.cs.vassar.edu/CES/

Languages on the Java VM: http://www.robert-tolksdorf.de/vmlanguages.html

Sage: http://modular.math.washington.edu/sage/

