UNIVERSITY OF MURCIA

International Journal of English Studies

IJES

www.um.es/ijes

Desiderata for Linguistic Software Design

GREGORY GARRETSON*
Boston University

ABSTRACT
This article presents a series of guidelines both for researchers in search of software to be used in linguistic analysis and for programmers designing such software. A description of the intended audience and the types of software under consideration and a review of some relevant literature are followed by a discussion of several important considerations in evaluations of a project's software requirements. The main section of the article then presents a series of guidelines or desiderata, grouped thematically, for the design of software for linguistic analysis. These describe goals and principles of interest both to programmers designing software and to researchers deciding on requirements or evaluating tools. The article closes with an exhortation to researchers and developers to communicate closely during the software design process and to understand the compromises that are necessary.

KEYWORDS: corpus linguistics, linguistic analysis, software, tools, programming, software design, software-aided, computer-aided, guidelines, best practices.

**Address for correspondence*: Gregory Garretson. Östanbäcksgatan 1, 87131 Härnösand, Sweden. Tel: +46-611-236-00. E-mail: gregory@bu.edu

I. INTRODUCTION

This article presents a general discussion of goals to strive for in designing software for use in linguistic analysis. It is intended primarily for two types of readers: researchers in linguistics or a similar field who require software for a research project, and programmers creating such software. Occasionally, an individual will embody both types, but usually, researchers will need to contract others to write their programs if they cannot find appropriate ready-made software. In such a case, there is a need for good communication between the researcher, who relies on the programmer to develop the software, and the programmer, who relies on the researcher to specify its functionality. As a linguist and a software developer, I myself am keenly aware of the differences between these two perspectives, and of the issues that should be—but often are not—discussed. This article is an attempt to promote engaged discussion of just how software for linguistic analysis (shortened hereafter to *linguistic software*) should be expected to perform, and how this can be achieved.

Specifically, two types of guidelines are offered. Section II provides advice for researchers getting started on a research project about how to go about determining whether the software they need is already available, and how to proceed if it is not. Section III presents a series of "desiderata" that set out principles for programmers to follow in designing linguistic software. These are also intended for researchers, however, because they provide a good basis for deciding what to look for, or what to ask for, in such software.

For purposes of illustration, reference will be made at a few points to a suite of linguistic software called Dexter, which I have designed. This and all of the other software mentioned is described in the Appendix; therefore no URLs will be given in the text.

I.1. Corpus linguistics

Computers are ubiquitous in research and academia nowadays. General tools such as word processors and web browsers are in such common use that hundreds of thousands of hours of work have gone into the development and refinement of the most popular programs. However, the situation is very different for the more special-purpose tools needed by many researchers. It is not too difficult to find a good statistics program, but finding good software for conducting discourse analysis is rather more difficult. Linguists have highly specialized research needs but are few enough in number that they do not constitute a terribly lucrative market. Therefore, while there are a number of good programs currently available (see the Appendix for more information), it sometimes happens that research teams must undertake or commission the creation of their own tools.

It used to be the case that linguists who wanted to base their research on large amounts of empirical data faced enormous obstacles, given the difficulty of amassing, organizing, and analyzing the data required. Over the past two or three decades, however, technology has gradually made it possible for huge corpora to be created and queried with

relative ease and sophistication. The creation and use of linguistic corpora—although generally not an end in itself—has become an area of such activity that the term *corpus linguistics* has come into currency. For the purposes of this article, we may say that anyone using an electronic corpus or database of language data for research or teaching is participating in corpus linguistics.[1]

The discussion below is generally oriented toward research projects in academic settings, where the goal is not to profit from the sale of software, but simply to enable some research to be carried out. In addition, anyone using software for applications in Computer Aided Language Learning or similar fields may well find the discussion to be relevant.

I.2. Previous work

Although a great deal has been written about software design in general, relatively few authors have discussed the specific requirements of the sorts of software considered here. This section presents a few works that address aspects of this specific area.

Alexa & Zuell (2000) review the software that was available as of 1998—now a decade ago—for "text analysis", though the programs they review are not those most typical of corpus linguistics. They discuss the often-cited dichotomy between "qualitative" and "quantitative" software packages, pointing out that there is often no sharp dividing line between the two, and focus instead on areas of functionality common to almost all such programs: data import and management, coding and characterization schemes, exploration operations, and export operations. In their review of fifteen different programs, they note that no two of these programs use a the same format for data encoding and storage, and none of them uses a standards-based format (such as SGML or XML), thereby making data exchange and data preservation difficult.

Where coding schemes are concerned, they point out that in some cases, the categories are fixed and determined by the maker of the software, whereas other programs allow the user to specify coding schemes, with varying degrees of complexity and flexibility. Even the unit that is to be coded (word, sentence, line, paragraph, etc.) varies greatly from one program to the next. The programs reviewed tend to allow either automated coding or manual coding, but not a combination of the two. While some of the programs include analytical linguistic operations such as lemmatization, Alexa & Zuell suggest that it would be useful for such operations to be included in a modular fashion, such that researchers may use only those they require, or may provide their own analyses. They also point out that where exploration (searching) is concerned, programs tend either to focus on an examination of the text itself or of the user-added codes, but not both. They call for developers "to aim for an open and extensible model for text analysis software" (2000:318) such that functionality may be added to programs as new needs arise and as new capabilities emerge. They also call for standards of data exchange, such that researchers may work with the strengths and weaknesses of different programs by using each one for the most appropriate tasks.

The type of software most commonly used by corpus linguists is the concordancer; Wiechmann & Fuhs (2006) is a review of ten such programs available as of 2006. Concordancers are typically used for searching the text of a corpus and displaying lists of the tokens found (typically called *concordance lines*), usually in the Key Word in Context (KWIC) format, although many of them offer considerably more advanced functionality. The review covers the most notable features as well as the technical limitations of the programs, which were tested using a range of corpus sizes and target frequency. They make the point that where large data sets are concerned (such as 100,000,000-word corpora), memory management becomes a critical consideration; all of the programs tested showed considerable limitations. Pointing out that "the more options the software offers, the less external work the user has to put into it" (2006:108), Wiechmann & Fuhs review the programs in terms of the types of analyses they allow and the ease with which the analyses may be performed. They note that many freely-available programs perform as well or better than the commercially available alternatives in many respects. Overall, their review underscores the fact that different programs have different strengths, in terms of power, range of features, and usability, and researchers are well advised to consider their needs carefully before choosing a program.

There are, unfortunately, areas of functionality in which concordancers perpetually show considerable weakness. Smith, Hoffmann & Rayson (2008) discuss one such area in detail. Specifically, they focus on the lack of good tools for annotating lists of concordance lines. Indeed, most concordancers were designed with searching in mind and not annotation (or only to a very limited degree), even though, as Smith *et al.* point out, it should be fairly obvious that such manual analysis will frequently be necessary. While it is possible to export the data to spreadsheet or database programs for annotation, this requires that the link between the tokens and the corpus be severed; it also means that the data may no longer be manipulated using the concordancer. Smith *et al.* discuss the possibilities currently offered by existing software for annotating concordance lines, as well as some options for using external programs, and mention some ways in which future tools might be designed to facilitate this important activity.

Turning to considerations of architecture, Davies (2005) argues for another aspect of linguistic software design, namely the use of relational databases. His remarks focus on the context of web-based concordancers, but his arguments apply equally well to any sort of software that needs to manage large amounts of data in a reasonably efficient manner. Davies discusses the usual trade-offs in software design between size, speed, and extent of annotation, and recommends the use of relational databases coupled with extensive indexing of the data as a means to achieve all three goals simultaneously. Two other benefits of the use of such an architecture are the ability to define highly complex queries without suffering much in processing time, and the possibility of linking in other resources such as semantic word nets and user-customized word lists. Davies also makes the observation that different languages require (or allow) different methods for processing data; for example, Spanish has much richer inflectional morphology than English, which allows part-of-speech tagging to be performed with greater ease and accuracy.

As a guide to best practices in corpus annotation, Leech (2005) offers many observations relevant to software design. In addition to part-of-speech tagging, Leech lists the following reasonably common types of annotation: phonetic, semantic, pragmatic, discourse, stylistic, and lexical. He notes, however, that "it is possible to think up untold kinds of annotation that might be useful for specific kinds of research"—and mentions disfluencies and learner errors as two interesting examples—which suggests that flexibility will usually be a very welcome feature in annotation tools. He goes on to list several desiderata for the application of annotations to a corpus, including exhortations that annotations should be removable, that annotation practices should be well documented, that widely accepted categories be used, and that annotation should be encoded using *de facto* standards (such as XML). Leech makes the point that the accuracy of a tool (such as a POS tagger) can only be measured if the annotation scheme is specified in detail, which provides another good argument for highly explicit documentation. And importantly, he makes the twin points that the annotation of a corpus should build on previous efforts and standards, and that improvements on these previous systems should nevertheless be pursued.

II. ADVICE FOR THE RESEARCHER

This section presents a discussion of some of the fundamental questions facing a research team or an individual researcher starting out on a project that will require some form of linguistic software.

II.1. Software requirements

When undertaking a project that will require some machine-processing of data, it is important to establish what the software requirements are—that is, what the software will be asked to do.

Asking oneself the following series of questions can be helpful: What is the *general* research question I want to ask? What *specific* questions must be answered in order to answer the general question? What will the answers to those questions look like? What kind of data, and how much, do I need to be able to ask those questions? Assuming I have the right data, what format is it in? How difficult would it be to modify the format? In case the right program is not available, what kind of access do I have to people who could design tools or modify my data for me? How much am I willing or able to pay for software or software design? How soon do I need to have these tools?

Knowing the specific research questions enables one to make a list of the specific analyses required. Knowing what these analyses are and what form the data is in should point to the kind of software needed. Finally, a clear picture of the resources and timeline involved will help to narrow the options.

II.2. Data format

It is important to bear in mind that every research project involves a combination of *software* and *data*. Both need to be appropriate to the research question—and to each other as well.

Finding the right data for a study is an important issue in corpus linguistics, and one that is too complex for discussion here. Suffice it to say that one either will make use of an existing corpus or database or will collect one's own data. In either case, the format of the data can be critical for several reasons. First of all, the format of the data—including the markup (see Section III.2.)—may be crucial for the analysis. For example, in order to ask questions about differences between male and female speakers, transcripts of spoken data will be required in which the sex of each speaker is marked up. This will probably take one of two forms: either each turn will be labeled with the sex of the speaker, or a list of speaker names with the sex of each will be created. Either method will work, but most likely not with the same tool. A program that searches only line-by-line would require the former, while a program that is able to cross-reference speaker data could make use of the latter. For example, Dexter, which is principally used to facilitate the manual annotation of written and transcribed data, includes a Converter program that converts documents to a standard XML format, which in turn enables many features in the annotation program. The Converter program is able to take a list of speaker data contained in the header of a document and include it in the XML version in such a way that the annotation program can filter searches by speaker name, sex, age, status, etc. However, such capabilities are still fairly rare in the sorts of software linguists currently use.

This underscores the fact that it is generally worth the extra effort to format one's data according to accepted standards in use in the wider research community, such as a TEI-compliant version of XML (see Section III.2). There are tools designed specifically to work with such data, which may save considerable time when it becomes desirable to convert, share, or otherwise manipulate the data.

It is also more generally true that different programs simply require differently formatted data. Although it is ideal for a program to be highly flexible in the kinds of input it can accept, in practice this is very difficult to achieve—the Dexter Converter program just mentioned includes nearly as many lines of source code as the annotation and search programs combined. In the absence of a tool that can perform the desired analysis with the data as it is, it becomes necessary to consider whether it is preferable to create a new tool or to reformat the data.

If the program required already exists, and the only problem is that the data is not in the right format, it may be worth it to hire someone to reformat the data. Although it can potentially be done manually (and cheaply) by a research assistant, this is often too laborious to be practical, and such manual reformatting can yield sloppy results. An alternative is to hire a programmer, not to create software *per se*, but simply to reformat the data. Converting data from one format to another is often surprisingly easy for someone with programming skills, and this could be the extent of the programming required in the project.

One point that is critical to keep in mind, whatever the data format used, is that the data should be as clean and as consistent as possible. Computers are extremely inflexible; they perform exactly the operations they have been instructed to perform, and no more. A human is able to read without problem a text that is rife with spelling errors, extra characters, and formatting inconsistencies, but a computer will often be thrown by these. Therefore, one should strive to make the data *clean*, in the sense that it should not contain extraneous characters that will confuse the program; typical offenders are line breaks, tabs, extra spaces, doubled punctuation, and unmatched quotation marks. One should also strive to make the data as *consistent* as possible. For example, if one were to mark up the language in use in a transcript sometimes with *language="eng"*, sometimes with *lang="english"*, and sometimes with *lang="en"*, very few programs would be able to identify these as equivalent. This is another area in which a programmer may be of use, since it is relatively easy to write programs that check for inconsistencies. This is also a good argument for using highly structured data formats such as XML, since these allow the validity of the structure and content of a document to be checked easily.

II.3. Using existing software

The most significant question in planning the computational aspect of a study is whether there exists ready-made software that can perform the necessary tasks, or whether it will be necessary to create or commission custom software.

It is generally preferable to use existing software, if the right tools can be found. There are a great many programs available; one should always perform a thorough search before concluding that the required tool does not exist. There are many places to look for information on existing tools; see the Appendix for some suggestions.

However, even if the right tool does not exist, that does not mean it is necessary to start from scratch. There are essentially four ways to make use of existing software code. The first is to use a program exactly as it is, possibly modifying one's data to suit it, as discussed above. The second is to modify an open-source program so that it fulfills one's needs. The third is to use the tools in a "toolkit" in series to achieve the desired results. The fourth is to write a program from scratch but include some code provided by others.[2]

An example of modifying an open-source program would be taking a syntactic parser designed for Spanish (and written, say, in Python), and modifying it to work for Portuguese. An example of taking advantage of a toolkit would be using the GATE framework (written in Java) to first part-of-speech tag a corpus and then extract all the references to named entities. A borderline case between using a toolkit and writing one's own program would be using the statistical programming language R to create graphs of word frequencies in a set of transcripts. Finally, an example of using others' code in one's own program would be writing a Perl program that allows a user to add annotations to XML documents without having to see the XML tags, making use of freely available code modules for XML parsing and for creating a GUI. See the Appendix for more on all of these resources.

The bottom line is that no matter what a program is required to do, there is already a certain amount of code out there that may be used as a foundation.

II.4. Problems with ready-made tools

Although it is certainly advantageous to use pre-packaged software, it may not always work well for a given project. There are four potential problems with ready-made tools that are worth bearing in mind while considering the options.

First, a given tool may not work as advertised. Especially if the program is not well documented or comes with an "as-is" disclaimer, it is important to scrutinize the results it produces to make sure that the tool does exactly what is expected. Second, all software has a learning curve; the more complex the program, the steeper the curve usually is. There are some tools that are so difficult to learn to use that it may not be worth the effort—it may be better to use another tool or even to do the work manually, if the data set is small and the analysis relatively simple. A third problem is that some tools work well with one type of data but poorly with other types. For example, tools designed to work with English data may not give good results with Swedish, much less Russian or Korean, data. Similarly, a tool will often require a very specific data format, as discussed above; it will then be necessary to evaluate whether it is worth the effort of reformatting to data to comply with its requirements. For example, the CLAN tools from the CHILDES system require all data to be in the highly specialized CHAT format.

The fourth and most subtle problem with using ready-made software is that, whenever one is not privy to the inner workings of the program, one is not certain what it is actually doing. This problem arises quite plainly with statistical software; all too often users click a few buttons and run a statistical test without really understanding what the test is measuring or even being certain that the test is appropriate. An advantage of open-source software is that it is possible for someone with knowledge of the programming language used to check the source code and ascertain what is actually happing. Of course, this is an uncommon situation. The general point to be made here is that it is incumbent on each user to know what the tools he or she uses are doing, which is a strong argument for using programs that are well-documented.

II.5. Creating custom software

In some cases, there will be no software that meets the requirements of the research project. Then the question becomes how to go about creating custom software. This is the scenario with which this article should prove most helpful.

Clearly, in order to create one's own software, it is necessary to have at least one programmer on the project. In the context of an academic research project, it is not too unusual to have one or more faculty members or students involved who have programming

skills. If not, it may be possible to collaborate with personnel from computer science or another technical department. It is often the case that research projects in linguistics and in natural language processing are being carried out simultaneously in different parts of the same institution with little or no communication between them. It may be worth exploring the possibility that two such groups' research interests overlap and that they may bring complementary skills to a project.

If all else fails, it is certainly possible to hire a programmer. The advantage of hiring professional consultants is that they may (in theory, at least) be held to a high degree of professionalism; the disadvantage is that they are often painfully expensive. Frequently, a student of computer science or a related field can be commissioned to do the work for significantly less. However, in such a case it is important to remember that fewer assumptions may be made about the quality and professionalism of the work done, not to mention adherence to timelines. In either case, the desiderata given in Section III might be used as a starting point for discussions of what is expected.

On occasion, it may be the case that a member of the research team who does not have many programming skills is willing to develop them. I believe that linguists should be encouraged to learn programming skills (for a discussion of the advantages, see Biber *et al.,* 1998:254-256), and can attest that it is possible for linguistic software to be developed by individuals without a background in computer science. If someone on a research project does begin learning to program, it is important that he or she receive the support necessary to develop sufficient skills to be useful to the project. The following are good ways to start: (a) attending classes or tutorials in programming, (b) reading good books on programming techniques in general and the chosen programming language in particular, (c) making use of the many resources on the Web for learning to program (including language-specific websites and discussion fora), and (d) finding others who are more skilled and are willing to occasionally share their expertise and provide guidance. It is also important to realize that this is hardly the fast track to developing a set of tools, so patience and flexibility will be required of the entire team.

III. DESIDERATA FOR SOFTWARE DESIGN

This, the main section of the article, presents a series of points to consider when designing linguistic software. These are directed to the programmer but are also of relevance to the researcher who expects to use the software. They are called "desiderata" here rather than "best practices", first because they primarily reflect the views of one individual and not an advisory body, and second because not all of them will be achievable for a given project. Rather, they are goals to aim for and principles to bear in mind while working on a software project.

The desiderata are grouped thematically under seven headings: *General design principles*; *Linguistic theory and linguistic data*; *The user interface*; *Documentation*; *Testing, debugging, and error handling*; *Search capabilities*; and *Performance*. These

sections, and the points within them, are ordered in a sequence that seems logical but is not intended to make any suggestions about relative importance. Each point starts with a sentence in bold type that sums up the desideratum. This is phrased as an imperative, simply because this is a more tractable form than English passives and modals. This imperative is followed by a brief discussion that elaborates on and motivates the guideline.

As much as possible, technical details and technical discussions are avoided. While this may reduce these points' immediate usefulness in some ways, it means that they apply to virtually any programming language or environment that might be used. The general terms "software" and "program" are used (instead of the more technical "application") to refer to any sort of software product, and the more familiar term "programmer" is used in preference to the more technical "software developer". Similarly, the researcher becomes the "user" when he or she starts to use a program.

III.1. General design principles

We begin with some general design considerations. Naturally, there is a large number of principles that guide good software design, and many books have been written about these. Only a few that are highly relevant to the design of linguistic software are mentioned here.

Choose robustness over complexity. Researchers use linguistic software to collect data for published research. The accuracy of the data is therefore of the utmost importance. Therefore, if you must choose between the careful design, testing, and debugging of a relatively small set of features on the one hand, and the hasty and untested addition of a large number of features on the other hand, choose the former. Doing three things well is better than doing ten things badly, especially when scientific research is at stake.

Leave enough time for development. Software development almost always goes over schedule. Even though it may seem urgent to start using the software, it is usually a bad idea to skip the testing and release it too soon. Prospective users should be prepared to wait until the software is certain to work. Better yet, leave more room in the project schedule for software development than you think will be needed.

Make the source code as modular as possible. Modularity is one of the most important principles of good software design, and especially in a context in which tools may be used in various ways, it pays to encapsulate a program's components into reusable units. Object-oriented programming is almost always the right choice for complex linguistic software. There are many excellent books on design patterns and principles of good object-oriented programming.

Do not reinvent the wheel. You will save tremendous amounts of time if you make use of the wealth of freely-available source code in most languages for performing most common tasks, and many uncommon ones. If your code is sufficiently modular, it should be reasonably simple to incorporate such code into your program.[3]

Make your source code available to others. Benefiting from code others have written is only possible when they share it. You should do the same. If you do not have a good reason to keep your code private, share it with the rest of the research community, using an open-source license (see Section III.4). In fact, it is often easier to get funding for a project if it is stipulated that the tools developed will be made publicly available.

Make your program flexible. Try to envision all of the different ways a user might try to use a tool, and maximize the number of questions that may be asked using the program. As I have argued elsewhere (e.g., Garretson & O'Connor, 2007), designing tools that can be reused is worth the extra time spent. Create a framework for adding features in the future as new uses for the program emerge. Of course, the best way to anticipate what users' needs will be is to develop the software in close consultation with those who will use it.

Accommodate a wide range of users. If you design a useful program, researchers in other countries, and researchers working with other languages, will want to use it. So develop software that is (a) able to handle data in various languages, (b) localizable (that is, may be translated to different languages), and (c) cross-platform. The support in most languages for Unicode text (see Section III.2) and cross-platform operation is steadily increasing, so these goals are no longer unrealistic.

Protect the user's data at all times. As Raskin (2000) puts it, "the user's data is sacred". Save all data to disk regularly. Ideally, do not keep any data in memory that is not simultaneously recorded on a hard disk or other drive. When appropriate, consider storing data on a central server, one on which regular backups are performed.

Choose clarity over cleverness. Programmers often find ingenious ways to solve problems. Unfortunately, these often lead to code that is impenetrable to other programmers. (Perl programmers in particular are notorious for this.) If you develop a program, try to avoid writing overly cryptic code, and document any ingenious solutions for the benefit of those who will maintain the code.

III.2. Linguistic theory and linguistic data

Linguists have special needs when it comes to program functionality and types of data. Here are some principles relating specifically to linguistic data.

Do not build linguistic theory into the program any more than is necessary. Sinclair (2004:190), *inter alia*, has issued many cautions about the dangers of incorporating a theoretical framework at too early a stage of data analysis. Even as common a practice as part-of-speech tagging has the tendency to oversimplify the data in the eyes of the researcher, leaving him or her blind to many patterns that do not neatly fit the framework imposed. As much as possible, tools should allow different theoretical perspectives to be taken. For example, a concordancer that can handle part-of-speech tags should be able to work with various taxonomies of tags. Make sure to document well any theoretical systems or annotation schemes that your tools do implement (see Leech 2005 for more on documenting annotations).

Do not gloss over complexities in the data. Even a simple count of words requires theoretical decisions about tokenization: How many words are *book-club* and *book club*? What about *isn't*? What about *−1.5*? Such issues need to be addressed in the design of the software, and the solutions should be documented so that users know what they are getting. When it makes sense, allow users to change these settings, though see also the point about "sensible defaults" in Section III.3. A good example is the "Token Definition" options in the AntConc concordancer. Also, if your tool provides statistical analyses, make sure to document how those statistics are calculated, especially when there is more than one possible formula. A good example is the WordSmith Tools documentation, which describes the several different measures of collocation strength calculated, and not only explains them in layman's terms, but gives the formulae as well as references to their published sources.

Separate markup from annotation. *Markup* is extra information added during the creation of a text. It supplies information that would be lost otherwise, as in "Bob called John a Republican, and then <emph>he</emph> insulted <emph>him</emph>", where the <emph> tags mark prosodic emphasis, in this case resulting in a switching of the pronouns' antecedents. *Annotation*, by contrast, is a record of a theoretical analysis of the data. A rule of thumb is that markup records things that are indisputable, while annotation does not. Part-of-speech tags, for example, are often argued to be annotation, since they are dependent on a theoretical system, with different researchers using different systems. Therefore, whenever possible, physically separate annotation from markup. If possible, keep the marked-up data in one file, and the annotations in a separate file; this is known as *stand-off markup* or *stand-off annotation* (Carletta et al., 2005) and is best accomplished using XML (see below in this section) or a relational database (Davies, 2005). Stand-off markup allows several different analyses to be made of the same data, kept in separate files, and queried either separately or together. At the very least, make sure to maintain a version of the data without any annotations.

Allow users to supply their own analytical categories. If your program allows only a fixed set of categories, it will limit analyses unnecessarily; different users may find different uses for the same tool. Also, any thorough analysis of data is likely to result in the creation of new categories. Charles Fillmore (1992:35) has said, "[E]very corpus that I've ever had a

chance to examine, however small, has taught me facts that I couldn't imagine finding out about in any other way". From this perspective, it seems likely that categories will continue to arise during the examination of a corpus, so software should support adding and changing categories as the exigencies of the analysis demand.

If a specific data set has been used to train a tool, describe it. Some tools, such as POS-taggers and parsers, need to be trained on a large data set, which will have certain characteristics. Each user needs to be able to judge whether his or her data is sufficiently similar to the training data to produce acceptable results. For example, a tagger trained on *Wall Street Journal* text will be likely to produce poor results on a corpus of spoken language data.

Make as few assumptions as possible about the nature of the data. The usefulness of a tool correlates in part with the variety of data it can be used with. For example, a great deal of linguistic data (spoken data, written learner data, and many other kinds) does not strictly follow standard orthography, so a reliance on punctuation rules could be a bad idea. Also, programs should not expect data in only one language; even a corpus of English will contain a large number of foreign words, in the form of loanwords and proper nouns.

Allow for different data formats. Related to the previous point, being flexible in terms of data format is highly desirable. No two research projects are likely to have identically formatted data. Although it is reasonable to expect a certain cleanliness in the data, the flexibility of your tool will determine its overall usefulness. If possible, allow for the use of data with no markup at all, with POS tags, and with XML or SGML markup. Also, allow the user to specify how different parts of a document should be interpreted. For example, how is the document header formatted? In the case of spoken data, how are speaker turns indicated? The Dexter Converter program, mentioned in Section II.2, allows the user to specify a great deal about how the markup of a document is to be interpreted: for example, how the speaker of a line in a transcript is marked, how overlapping speech is indicated, how pauses are marked, whether there are no tags, tags with angle brackets, tags with square brackets, etc.

Use Unicode. The importance of making programs useful for data in different languages has already been pointed out. The simplest way to achieve this is to use Unicode, which is an international standard character set that covers all written natural languages. Nowadays most programming languages have good Unicode support. Note that you will also have to choose a character *encoding*; whereas a character set is an abstract list of characters, an encoding is the way these characters are represented in binary form in files. The best general-purpose Unicode encoding (at least for non-Asian languages) is UTF-8 (McEnery & Xiao 2005). Make sure to use word-processing tools that can handle the encoding you choose.

Make use of open standards for data storage. The use of idiosyncratic or proprietary data formats, especially binary ones that leave the code inaccessible using more than one type of program, is a dangerous and unadvisable practice. Instead, XML (a descendant of SGML) is

quickly becoming the standard format for storing data in a well-defined, flexible, and safe way. Because its syntax is well-defined, it can be read and processed automatically by computers. Because its semantics is flexible, it can be used for a wide variety of types of data. Because it is simply plain text, it can be accessed and changed with ordinary text editors, as well as with specialized programs. There are many guidelines, such as those of the Text Encoding Initiative, that give extensive advice on how to use XML.

III.3. The user interface

The programmer who develops a program is intimately familiar with all of the various parts of the software. By contrast, a researcher who uses the program sees nothing except the *user interface*. The usability—and indeed, the use or non-use—of a program is dependent on the quality of its user interface.

Design the interface for the least technical user. Nothing is more frustrating that an interface that demands arcane knowledge or is simply confusing. A user may reasonably be expected to learn a few things in order to use a tool, but that number should be kept to a minimum. A more technical user will not be hindered by a well-designed interface, but a less technical user *will* be hindered by a badly-designed one.

Provide a set of options rather than requiring the user to know what to type. Any time there is a limited number of options, it is better to list the options and allow the user to choose one, rather than to expect the user to have memorized all options. This leads directly to the next point…

Use a graphical user interface rather than a command-line interface. Few things are more intimidating to a non-programmer than a terminal-style interface: a blank screen with a blinking cursor. While UNIX and Linux devotees may be quite comfortable with the command line, it is important to recognize that such an interface depends on a thorough familiarity with the tools, which is quite expensive to obtain. A graphical user interface (i.e., using windows and a mouse), or GUI, is not only easier to use, but will be more familiar to most users. It is undeniably more work for a programmer to design a GUI, but if it allows a far greater number of people to use the program, the gains will probably outweigh the costs.[4]

Respect existing conventions of GUI design. Make good use of the various components available. For example, use radio buttons for options that are mutually exclusive and checkboxes for those that are not. Use drop-down listboxes for choosing a single item from a short list, and scrolling listboxes for choosing more than one item. Using the sorts of menu systems that users will expect will flatten out the learning curve. Do not violate conventions unless you have come up with a clearly superior design.

Provide functionality the user will expect. There are many types of low-level functionality that most users will expect in a program, such as cutting and pasting text, switching freely between windows, using keyboard shortcuts, and having an "undo" option. Such functionality is important and relatively inexpensive to provide.

Make semiotic choices carefully. In recent years, interface designers have noted that icons are not usually as user-friendly as simple text labels, because it is very difficult to come up with pictorial symbols to represent abstract processes (Raskin, 2000). Although there are certain well-entrenched icons (such as those for "save" and "print"), before you use symbols, make sure they clearly convey to test users what they are supposed to convey. Similarly, try to make sensible choices about the layout and look of your program's interface; for example, a button that initiates an important function should be highly salient, but not so large that it doesn't look like a button.

Make the interface customizable. In the case of components that show large amounts of text, allow the font and the size of the text to be changed. Also, make the various components of the GUI resizable whenever possible, since different users have different-sized monitors. If a user needs to set several options before using the program, those options should be saved for the next session; it may also be desirable to allow the user to store different configurations of settings in different profiles.

Make sure the most common operations are the easiest to perform. In most programs, there are certain central actions that are performed often; the program should be designed around facilitating these actions. The best way to do this is usually to supply keyboard shortcuts. A reasonably experienced user can work much more quickly using the keyboard for common operations rather than the mouse. This is especially true if a sequence of operations needs to be repeated; pushing three keys in a row is vastly preferable to scrolling through three menus over and over.

Minimize repetitiveness. Computers' main value is in automating and speeding up repetitive operations. Make sure to minimize the extent to which repetitive actions are demanded of the user. One example of this is creating macros, in which a single trigger starts a predefined series of operations. Another example is batch processing of files, whereby the same operation is performed on an entire list of files. Yet another example is saving options to a configuration file to be loaded when the program is run.

Supply sensible defaults. Do not force the user to make a choice when a sensible default option can be supplied. Very often, a number of choices must be made before a program can be used. For example, the user may be able to configure what counts as a word, what will be done about case-matching, and how the output will be formatted. In most cases, there is a most likely choice for each option; make this the default. The user should be *allowed* to make choices, not *forced* to.

Make it easy to change options. If the user is allowed to make choices, it should not be excruciatingly difficult to do so. A well-designed interface should enable any setting to be changed quickly and with only a few mouse clicks or key presses. This means that menus should not nest too deeply. Also, making the user manually alter a configuration file on the hard drive may be better than not allowing any configuration at all, but it is not nearly as good as allowing changes through the user interface.

Tell the user what the performance consequences of various options will be. If choosing an option will slow down the program, state that clearly. For example, if turning on regular expression matching in searches will noticeably slow down searches, alert the user so that he or she will not use that option when it is unnecessary. Similarly, if there is an option that a user would choose *only* for performance reasons, make this clear, and explain what its effects on the program's function will be. A checkbox that says "Fast search" or the like is insufficiently informative.

Keep the user informed about what is happening at all times. Whenever an operation takes more than a second or two to complete, alert the user to the state of the program and what it is doing. Otherwise, the user may assume that it has frozen and quit the program, causing loss of data. The status bar at the bottom of a window is well suited to this purpose. Similarly, for operations that predictably take some time, include a progress bar to show the percentage of the operation that has been completed.

Present results in stages. If your program generates a series of results, especially numerical ones, it is good to present them hierarchically, giving an overview first, and then allowing the user to choose to examine the results in more detail or from different perspectives. For example, in a program that showed the distribution of parts of speech in a corpus, it would be preferable to show the major classes—noun, verb, adjective, etc.—first, and give the *option* of comparing all of the subtypes of nouns—singular uncountable nouns, plural countable nouns, etc.—rather than showing all of this information at once. Another application of this idea is to allow the user to filter a set of search results progressively. For example, one might first find all sentences containing the string *'s*, and within those, eliminate the ones containing *it's*, *he's*, *she's*, etc.

Give the user access to logs and configuration files. Especially if you do not expect to be personally maintaining the software forever, you should make it possible for the user to figure out what went wrong by consulting the program logs. Similarly, if program configuration information is saved in a file on the hard drive, let the user know where that file is (especially if something cannot be changed via the user interface).

Allow the user to decide where to place application and data files. Especially with a program that runs on various platforms, it is foolhardy to make assumptions about the file

system on the user's computer, and overly intrusive to insist on the program being installed in a particular directory. This is one thing that the user should be able to choose when setting up the software. It is even more important to allow the user to choose where the data will be stored. Not only does the data belong to the user, but it is much more likely that the data will be backed up if it is stored in the user's directory, and not with the program code or hidden elsewhere in the file system.

Make it easy to install and update the program. A difficult installation is the single best way to discourage users. If possible, use an auto-installer to simplify the process for the user. It is similarly important that the user be able to install new versions of the program as they become available. You may wish to have the program automatically check for updates, although it is usually best to allow the user to choose whether to install them. Seriously consider the advantages the World Wide Web can confer. A good example is Java Web Start, a framework that allows one-click installation and automatic updates of Java programs.

III.4. Documentation

Documentation is often given short shrift by programmers who see it as something extra and optional. However, as many a user will attest, good documentation is a critical part of any software—a program that "works perfectly" but is impenetrable to users will do no work at all.

Document right from the start. Leaving the documentation for last is a mistake, especially because you run the risk of never writing it. Instead, document features as they are added. This is easier and leads to more complete descriptions.

Err on the side of being over-informative, rather than under-informative. Although too much detail may intimidate users, lacking important details is even worse. Aim to make the documentation sufficient to fully train a new user. You may wish to make use of images or video display technologies for demonstrating the basic functions of the program.

Write the documentation in a way the reader will understand. It is important to distinguish user-oriented documentation, such as the user manual, from programmer-oriented documentation, such as the technical manual. These need to be written in very different styles, as they assume very different knowledge and goals in the reader. Make sure that all terms are explained before they are used or can easily be looked up.

Make it easy to find documentation on a given topic. Provide capabilities for searching the documentation, both by topic and by full-text search; many languages provide modular help-program code that is easy to integrate. Hyperlinked HTML documents make an

excellent format for user documentation, and can be put on the Web as well. Another good resource is the "tool tip", a text bubble that appears when the user holds the cursor over an option for a few seconds.

Provide clear contact information. Make it easy for a user with questions to figure out who to contact. Make sure to distinguish the individuals responsible for the software from those responsible for other aspects of a project, and make it clear what kinds of support the user may (or may not) expect.

Make the terms of use clear. Specify the conditions under which others may use your software, for example whether it may freely be used for academic research, and whether it is permissible to include it in packages sold for profit. Provide an appropriate license, such as the GNU General Public License, and stipulate that this must be distributed with the software. For more information on open-source licenses, see St. Laurent (2004).

Publicly announce all bugs and bug fixes. Maintain a publicly available list of known bugs, and a change log (list of changes by version) so that users will know when bugs have been fixed. Announcing known bugs will allow researchers who use the software to evaluate whether their results need to be checked.

Document the code for other programmers. Source code files should have both overview documentation and line-by-line comments that explain what each block of code is doing. This is important not only for others but for yourself in the future.[5]

Provide a framework for contributing code to the project. It very often happens that the first person to work on a software project is not the last person to work on it. Even if you are starting out alone, it is best to assume that others will become involved at some point and facilitate collaboration as much as possible. For example, use a revision-control system such as Subversion; these allow various programmers to contribute code without interfering with each other's work, merging files as needed and keeping backups of all versions. Also, provide a set of coding conventions that everyone contributing to the project should follow.[6]

III.5. Testing, debugging, and error handling

Software that runs perfectly in all cases as soon as it is written is in all probability non-existent. Even the world's largest software companies ship software that has bugs. Debugging is a fact of life; testing a program to remove most bugs *before* users ever see them is even better. A thorough testing regimen requires discipline but makes for far more robust software, which saves time later on.

Test the program as you develop it. The best way to create robust software is to continually write tests to reveal bugs while the code is being written. Using a "test harness" makes this easier, and allows whole suites of tests to be run and rerun easily.
Test the code every time significant changes have been made to it. It often happens that fixing one thing breaks another. If you have a set of tests, you can run them every time you add a feature or fix a bug, to make sure that nothing new has been broken. Better yet, before you make a change, add a test that fails before the change and succeeds after the change. That way, if future changes ever break that feature, it will be detected.

Use beta-testers. Once the program is functional, but before it is finished, have a small group of users try it out. Their goal should be to find ways to break the software, so that it can be improved and made robust enough for real use. Ideally, have them try it with the actual data it will be used with.

Provide a set of test data with the program. Often, the best way for a user to become familiar with a program is to have a simple set of data to try it out on, especially while reading the documentation. This data set can serve a second purpose as well: A user encountering a problem with the software can try to replicate the problem with the test data, which acts as a known quantity. If the program runs fine on the test data, the next step is to look carefully at the user's data to try to locate the problem.

Report errors appropriately. Report errors to the user in a way that will be meaningful to the user, and to a log in a way that will be useful to the developer. I have seen instances of a dialog box with the message "An error has occurred", which is unacceptable. A user presented with an error message should have some idea of what went wrong, and should have enough information to submit a useful report to the programmer. Ideally, make it possible to copy and paste the text of the error message. On the other hand, if an error is fairly trivial, consider whether it is even necessary to report it to the user; error messages are alarming and can erode confidence in the software.

Make it easy for the user to report problems. Establish a clear channel for reporting problems and possible bugs. The user should know what information to include when reporting a problem and also know how to find the log and send it alongside the report.
Document error messages. The user documentation should include an explanation, for each error message one may encounter, of what it means, what to do about it, and how to avoid it in the first place.

III.6. Search capabilities

Because searching a set of textual data for patterns is one of the most common functions in linguistic software, the following points apply specifically to this area.

Weigh complexity against speed. There is generally a tension between having fast searches and having many search options. Where to find the right balance will depend on the purpose of your program and the data you intend to use it with. Make sure to test search functionality on full-sized data sets with real data.

Allow regular expressions. Most programming languages allow an extraordinary level of complexity in textual searches through the use of regular expressions, as when "\d" is used to represent any digit, "[aeiou]" is used to represent any English orthographic vowel, or "\s+" is used to represent any combination of spaces, tabs, and line breaks. For the searching of text, giving the option of using regular expressions is a valuable service to the user. However, this lays on the programmer the twin burdens of explaining the regular expression language to the user and making sure that the code will work as advertised. If you believe your program may give false results with regular expressions enabled, see the very first point above.

Use Unicode in regular expressions. Modern programming languages now include the capability of using Unicode properties in regular expressions. This means that one may write "\p{Lu}" to represent an uppercase letter *in any language*, rather than the old-fashioned "[A-Z]", which only works for English. If you want your program to work for data in various languages, using Unicode in regular expressions is a necessity.

Thoroughly document search options. It is of the utmost importance that the user understand what his or her search terms actually mean. If you allow the use of unadulterated regular expressions, you should provide, minimally, a list of the symbols allowed and a pointer to where to find more information. Make certain to include examples of search terms and the sorts of things they will match. You may decide instead to allow a subset of regular expressions, such as the "wildcards" * and + familiar from word processors. In this case, explain clearly what these represent. If you provide options such as "make search case-sensitive", make sure to explain what they do and how they may interact with other options.

Allow the user to supply lists of search terms. It significantly increases the power of a program, and cuts down on repetitiveness (see above), if the user is allowed to supply a list of search terms. One implementation of this is to chain together search terms using the Boolean operators AND and OR, as in a search for "maybe OR perhaps". Another implementation is to allow the user to create a text file with a list of search terms, which the software then opens and searches for in sequence, as is possible with WordSmith Tools (Scott, 2004).

III.7. Performance

We end the list of desiderata with a few slightly more technical points regarding software performance. The goal is to encourage programmers to think about certain issues that are somewhat more advanced but are nonetheless of great importance in the overall functioning of a program.

Use system resources judiciously. You must be prepared for the very likely case that a user will have a less powerful computer and several programs running simultaneously. It is important that your program not require more than a reasonable amount of memory and hard disk space. Wiechmann & Fuhs (2006) found that all of the concordancers they tested ran out of memory while performing certain tasks. You may wish to allow the user to adjust the amount of memory used by the program, though be sure to explain the ramifications (no pun intended). Meanwhile, in designing the program, beware of memory leaks and avoid them at all costs.

Cache data when it can improve performance. It sometimes helps to store data that is calculated frequently (or looked up in a database frequently) in a structure where it may be accessed again with less overhead. In the extreme case, this principle points to indexing all of the data that will be searched, so that the raw data need not be searched over and over (as described in Davies, 2005). This is how Web search engines are able to resolve queries with exceptional speed.

Benchmark, profile, and refactor your code to improve performance. *Benchmarking* means comparing two ways of performing a task by writing test code that performs the task thousands of times each way and compares their speed. *Profiling* involves using tools to examine your program while it is running, to determine which data structures use the most memory and which code takes the most time to execute, in order to see where improvements are most needed. *Refactoring* is rewriting code in such a way that it produces the same output in a more optimal manner. Such practices can markedly improve the performance of a program.

Use programming aids to increase your own performance. No matter what language you program in, using an Integrated Development Environment (IDE), such as Eclipse, will increase your own performance. An IDE is essentially a text editor with specific knowledge of a programming language, enabling syntax checking, auto-completion, automatic compiling, and debugging. A good IDE can dramatically speed up development, reduce errors, and even make sophisticated code refactoring suggestions (as in the case of IntelliJ IDEA).

IV. Conclusion

This article has presented guidelines both for researchers looking for linguistic software and for programmers creating such software. It is to be hoped that the desiderata given above may serve as a starting point for discussions between those developing software and those who will use it, as well as a set of principles for programmers to keep in mind while working.

It seems appropriate to close with an exhortation to both the researcher and the programmer to consider each other's perspective, as this is a theme that runs through much of the discussion above. The programmer designing software for linguistic analysis must understand what the needs of the researcher are and how these needs may vary from one individual to another. At a more fundamental level, the programmer must understand what constitutes a productive and enjoyable user experience for both novice and accustomed users.

At the same time, the researcher who needs such software should appreciate the difficulty of delivering many of the features called for here. Requiring more functionality, more flexibility, and more safeguards requires more work. Unfortunately, it is probably unrealistic to expect all of the desiderata listed above to be fulfilled by a given project, unless time, money, and skill are available in abundance.

Therefore, it is advisable, when setting out to create software tools, to have frequent and open communication among everyone with a stake in the software, including those designing it, those who will ultimately use it, and those holding the purse-strings. All should be prepared to identify what they require, what they would like to have, and what they are willing to sacrifice.

As a final note, I would like to emphasize that it is indeed possible for small research teams to develop their own software. I hope to have suggested how such an endeavor might be undertaken, how it might be facilitated, and how its chances of success might be improved. I welcome anyone encouraged to embark on such a project to the stimulating world of linguistic software design.

ACKNOWLEDGEMENTS

I would like to thank Annelie Ädel and Sabine Bartsch (as well as the two anonymous reviewers) for very insightful comments on drafts of this article, which led to many improvements. More generally, I am grateful to all of the linguists, programmers, and software users who have helped me to develop in the dual roles of linguist and programmer—especially Cathy O'Connor, who has championed my progress in both roles.

NOTES

[1.] The discussion in this article excludes most types of tools used in the wider field of *computational linguistics*—which includes applications such as speech recognition, speech generation, machine translation,

etc.—focusing instead on the types of research more commonly carried out within the more restricted domain of *corpus linguistics*.

[2.] It is even possible for a program to make use of code written in a different language. For example, because code written in the language C tends to run very efficiently, there are code libraries written in C with interfaces allowing code in more high-level and user-friendly languages to make use of them.

[3.] One caveat: You do not want to risk your program being crippled when some third-party software becomes unavailable or changes dramatically. Therefore, make sure to distribute all required code with your program—within reason, of course: some programs are needlessly distributed along with the entire Java Runtime Environment, which is usually already on the user's computer and is easy to obtain if not. The result is hard-drive clutter and excessive download times.

[4.] If you must use a command-line interface, at least make sure to design it carefully. For some helpful suggestions see Conway (2005:299-317).

[5.] Eagleson's Law states that any code of your own that you haven't looked at for six or more months might as well have been written by someone else.

[6.] While involving others can be highly beneficial, bear in mind that adding programmers to a project does not always speed up the project. Every time the necessary amount of communication in a project increases, it reduces its efficiency—see Brooks (1975), a classic of software management, for an analysis.

REFERENCES

Alexa, M. & Zuell, C. (2000). Text Analysis Software: Commonalities, Differences and Limitations: The Results of a Review. *Quality and Quantity* 34: 299–321.

Biber, D., Reppen, R., & Conrad, S. (1998). *Corpus linguistics: Investigating language structure and use.* Cambridge: Cambridge University Press.

Brooks, F. P. (1975). *The mythical man-month: Essays on software engineering.* Reading, Mass.: Addison-Wesley Pub. Co.

Carletta, J., McKelvie, D., Isard, A., Mengel, A., Klein, M., & Møller, M. B. (2005). A generic approach to software support for linguistic annotation using XML. In G. Sampson & D. McCarthy (Eds.), *Corpus linguistics: Readings in a widening discipline*. London: Continuum.

Conway, D. (2005). *Perl best practices.* Sebastopol, CA: O'Reilly.

Davies, M. (2005). The advantage of using relational databases for large corpora: Speed, advanced queries, and unlimited annotation. *International Journal of Corpus Linguistics* 10: 307–334.

Fillmore, C. (1992). "Corpus Linguistics" or "Computer-aided armchair linguistics". In J. Svartvik (Ed.), *Directions in corpus linguistics: Proceedings of Nobel Symposium 82, Stockholm, 4-8 August 1991*. Berlin: Mouton de Gruyter.

Garretson, G. & O'Connor, M. C. (2007). Between the Humanist and the Modernist: Semi-automated analysis of linguistic corpora. In E. Fitzpatrick (Ed.), *Corpus linguistics beyond the word: Corpus research from phrase to discourse.* Amsterdam: Rodopi.

Leech, G. (2005). Adding Linguistic Annotation. In M. Wynne. (Ed.), *Developing linguistic corpora: A guide to good practice*. Oxford: Oxbow Books. 17–29. Available online from http://ahds.ac.uk/linguistic-corpora/ [Accessed 2008-06-28].

McEnery, A. & Xiao, R. (2005). Character Encoding in Corpus Construction. In M. Wynne. (Ed.), *Developing linguistic corpora: A guide to good practice*. Oxford: Oxbow Books. 47-58. Available online from http://ahds.ac.uk/linguistic-corpora/ [Accessed 2008-06-28].

Raskin, J. (2000). *The humane interface: New directions for designing interactive systems.* Reading, MA: Addison Wesley.

Sinclair, J. M. (2004). *Trust the text: Language, corpus and discourse.* London: Routledge.

Smith, N., Hoffmann, S., & Rayson, P. (2008). Corpus Tools and Methods, Today and Tomorrow: Incorporating Linguists' Manual Annotations. *Literary and Linguistic Computing* 23, 163–180.

St. Laurent, A. M. (2004). *Understanding open source and free software licensing.* Sebastopol, CA: O'Reilly.

Wiechmann, D. & Fuhs, S. (2006). Concordancing software. *Corpus Linguistics and Linguistics Theory* 2(1): 107–27.

APPENDIX: A BRIEF LIST OF RESOURCES FOR FINDING AND CREATING LINGUISTIC SOFTWARE

Here is a list of selected resources for finding and creating software and learning about existing standards. The URLs given were current in June 2008.

General websites and mailing lists

David Lee's Bookmarks for Corpus-based Linguists is an excellent resource for finding both corpora and linguistic software:
http://devoted.to/corpora

The Stanford NLP Group has two useful pages of links to resources for computational linguists and corpus linguists:
http://www-nlp.stanford.edu/links/linguistics.html
http://www-nlp.stanford.edu/links/statnlp.html

In addition to having threads on a great many topics, the **Linguist List** also hosts the archives of two other relevant lists—the **Corpora List** and the **Corpus Linguistics and Language Teaching List**:
http://linguistlist.org/issues/index.html
http://listserv.linguistlist.org/archives/corpora.html
http://listserv.linguistlist.org/archives/cllt.html

Data format and encoding standards

The **Unicode Home Page** provides extensive information about Unicode:
http://www.unicode.org

Markus Kuhn's UTF-8 and Unicode FAQ for Unix/Linux is a more technical introduction to Unicode with an excellent explanation of UTF-8:
http://www.cl.cam.ac.uk/~mgk25/unicode.html

Peter Flynn's **XML FAQ** is a good place to learn about XML:
http://xml.silmaril.ie

The **Text Encoding Initiative** produces guidelines for the standardization of digital document formats. They also provide a **Gentle Introduction to XML**, which is a very good way to become acquainted with XML:
http://www.tei-c.org
http://www.tei-c.org/release/doc/tei-p5-doc/en/html/SG.html

The **E-MELD School of Best Practices in Digital Language Documentation** has more good advice on how best to format and store digital data:
http://emeld.org/school/index.html

The **GNU General Public License** is one of the most common open-source licenses for software, and a good example of such licenses:
http://www.gnu.org/licenses

Software tools mentioned in the text

Dexter is a suite of tools developed by the author for facilitating the manual annotation of linguistic data. Originally intended for working with spoken or written data one text at a time, the Dexter suite is gradually developing into a full corpus application. The Dexter Converter is a highly configurable, interactive tool for converting plain-text documents into TEI-compliant XML documents. The Dexter Coder works with these documents, allowing the application of annotations in any scheme of the user's design; these are displayed as colored bubbles overlaid on the text but stored as stand-off data in XML files. Dexter allows complex searches on both annotations and text, including regular expressions and filtering by speaker characteristics. Written in Java, Dexter runs on any operating system and is freely available:
http://www.dextercoder.org

WordSmith Tools, developed by Mike Scott, is probably the most commonly used corpus linguistic software. It is a suite of three tools: Concord is a concordancer that, in addition to providing Key Word In Context (KWIC) views, determines collocates, clusters, and dispersion of words through texts. WordList is a tool for creating lists of the words in a text or set of texts, for example to compare word frequencies. KeyWords compares word lists to determine the key words in a particular text or set of texts. WordSmith runs only on Windows and is available for purchase from Oxford University Press:
http://www.lexically.net/wordsmith

AntConc is a concordancer developed by Laurence Anthony. In addition to KWIC views, it provides concordance plots, clusters, collocates, word lists, and keyword lists. Its functionality is not quite as extensive as that of WordSmith, but it is freely available and runs on most operating systems:
http://www.antlab.sci.waseda.ac.jp/software.html

For a review of several more concordancing programs, see Wiechmann & Fuhs (2006).

CLAN is a set of tools developed by Leonid Spektor as part of the CHILDES system. CHILDES involves a database of child (and other) language transcripts, a transcription format called CHAT, and a set of tools for analyzing the data (CLAN). The CLAN tools

allow the user to perform many types of queries on a set of transcripts from the database via a command-line interface. For example, CLAN can calculate mean length of utterance or the frequency of different phonemes for different speakers in a set of documents. A user may freely use CLAN with his or her own data, provided it is in CHAT format:
http://childes.psy.cmu.edu/clan

GATE is an open-source framework for developing Java natural-language-processing software. It includes a large number of built-in tools, for example for POS tagging and named-entity recognition. It also includes an IDE that facilitates combining and modifying existing tools to create new tools:
http://gate.ac.uk

Java Web Start is a system for deploying Java programs via the Internet with as little as one click. When a user clicks a link on a web page to a special file, it instructs Java to download the program, install it, and run it. The program may then be run on the user's computer offline. JWS may be set to automatically download and install program updates when they are available:
http://java.sun.com/products/javawebstart

Subversion is an example of a *revision control system* (or *version control system*). Such a system allows several programmers to work on a project simultaneously without interfering with each other. Individuals "check out" source files as from a library, modify them, and check them back it again; the system merges different programmers' changes to the same file. All versions of all files are saved so that data is never lost and it is always possible to revert to a previous state. Subversion is free and open-source:
http://subversion.tigris.org

Eclipse is a popular Integrated Development Environment (IDE) for Java and other languages. It is freely available and works on many operating systems:
http://www.eclipse.org

IntelliJIDEA is an IDE for Java featuring extensive built-in refactoring intelligence, enabling it to provide suggestions for improving the user's code:
http://www.jetbrains.com/idea

Programming languages mentioned in the text

Java, Perl, Python, and (to a lesser extent) R are all programming languages that are in use by a number of linguists. It is beyond the scope of this article to compare them, but you may learn about them on their respective websites:
http://www.java.com
http://www.perl.org
http://www.python.org
http://www.r-project.org

