The Use of Corpora in Second Language Learning

by

Michael Colley

(Under the direction of William A. Kretzschmar, Jr.)

Abstract

Corpora can be a valuable resource for students learning a foreign language. Unlike most language resources, corpora show language as it is actually used in real situations. However, attempts to use corpora in second language learning have had limited success due to the fact that they use software designed for linguists rather than language learners. This thesis discusses the development and use of a program designed with the needs of language learners in mind. Upon entering a word, the program extracts phrases from the corpus based on words that most frequently collocate with the given word. This is useful for language learners because it allows them to see how the meanings of words are shaped by the surrounding context. Compared to programs used by linguists, it does a better job of hiding the subtleties of word usage and emphasizes the most frequent and therefore most useful characteristics of the language.

Index words: Corpus studies, second language learning, programming, applied corpus linguistics

The Use of Corpora in Second Language Learning

by

Michael Colley

B.A., Emory University, 2001

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Arts

Athens, Georgia

2004

© 2004

Michael Colley

All Rights Reserved

The Use of Corpora in Second Language Learning

by

Michael Colley

Major Professor: William A. Kretzschmar, Jr.

Committee: Don R. McCreary
Stephen Ramsay

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2004

Acknowledgments

Thanks to my committee members for their help with this program, as well as the thesis: Bill Kretzschmar, Don McCreary, and Steve Ramsay.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Students who are learning a widely-spoken second language generally have a lot of tools at their disposal: dictionaries, grammar books, phrase books, cassettes and CDs, reading materials, classes, and even computer software. One tool that has not yet made it into the mainstream, despite some experimental trials by researchers, is the use of corpora. There is no substitute for experience; in order to learn a foreign language, you have to be exposed to it, either through writing, spoken words, or (preferably) both. A corpus is like a large database of language experience. Unlike dictionaries, grammar books, and phrase books, corpora contain only attested language, language that was actually produced by somebody for some real purpose, not just for the purpose of illustrating a grammatical point or the use of a word.

The most useful feature of a corpus is that it is searchable electronically. By itself, a corpus is no more useful than a large collection of reading material. With computer software, however, a corpus becomes a valuable tool that linguists are increasingly using to do language research. The goal of this thesis is to design a program that will help make corpora an equally valuable tool for second language learners. Many researchers, such as Wooldridge (1991), Bernardini (2000, 2002), and Mair (2002) have already experimented with using corpora to help teach a foreign language. But there has not yet been any initiative to modify the software that linguists use to fit the needs of second language learners.

There are two major approaches to the use of corpora in language pedagogy. Traditionally, the most common approach has been to use corpora in the preparation of classroom materials such as readings or exercises. With rapid advances in computer memory and speed however, it has also become popular for teachers to give students direct access to corpora so they can investigate the language themselves. Silvia Bernardini (2000) has called this second approach the 'non-mediated' use of corpora, since the students interact directly with the corpus, rather than being presented with materials that the teacher has prepared using a corpus. The use of the term 'non-mediated' is a little misleading however. While it is accurate in the sense that the students are interacting directly with the corpus data, the corpus is still mediated by the software used to visualize it, in most cases a concordance. A corpus is simply a collection of texts, usually selected with some criteria in mind. What makes a corpus valuable is the ability to search through it and view its contents in different ways, such as via a word list or concordance. A truly non-mediated use of a corpus, that is reading it from start to finish, would not be a particularly useful or enlightening experience.

Programs used in conjunction with corpora are like a new pair of eyes that allow users to see things in a text that they would not have otherwise noticed. The text does not change, only the way it is presented to the user. In this sense, the presentation of data may be more important than the data itself. Texts have been around for centuries, but it is only with the ability to rapidly change the way they are presented that linguists have begun to see them in a different light. One implication of this is that there is no single best way to present the data from a corpus. The most common types of programs used with corpora have been word lists and concordances. The most popular type of concordance that linguists use is the key-word-in context (KWIC) concordance, which shows the word in interest (the keyword) lined up and centered, along with a certain span of words to the left and right of the keyword. An example is shown in Figure 5.1 on page 55.

Concordances that can be sorted by the words to the left or right of the keyword are particularly useful in noticing patterns of word usage such as frequent collocates. Software packages like WordSmith Tools offer a collection of simple tools to make a word list and a KWIC concordance along with numerous additional, often more complex tools for analyzing texts. WordSmith and similar programs were designed primarily for linguists, with features that are best suited to linguistic research. Although researchers such as Bernardini (2000, 2002) have been successful in using WordSmith and other programs in the classroom, the tools themselves were not designed for second language learning. It is not surprising then that such approaches often put the language learner in the role of a language researcher. While this may be a useful experience for some, it is not the only way that corpora can be used. The program created for this thesis will be designed specifically for second language learners, and thus will cater more to the goal of language learning than language research.

The program will focus on phrases rather than individual words. Corpus researchers like Michael Stubbs (2001) have argued that phrases, as opposed to words, are the main unit in which language is understood. In natural conversation and writing, words are seldom used in isolation. Thus the dictionary model of language, in which individual words have a list of definitions from which language users can choose, is misleading. Instead, corpus research has shown that most common words tend to collocate around a relatively small number of other words. This suggests that word choice is not completely open or random but is constrained by the context of adjacent words. This fact is reflected in collocation dictionaries like The BBI Combinatory Dictionary of English (Benson et al. 1997). The BBI dictionary differs from most phrase dictionaries in that it does not simply list common idioms in English like 'hit the bull's eye.' Instead, it lists phrases formed by frequent collocates, such as frequent verb-noun combinations (e.g. 'make a bet') and preposition-noun combinations (e.g. 'in advance'). These types of expressions are more important for second language learners than colloquial idioms since they tend to occur with much greater frequency. These are the types of expressions that this program will focus on.

Although most concordance programs allow users to search for phrases, the user must know in advance which phrases are important. For a second language learner, however, this is not a straightforward decision. For example, there is no logical reason we *take* advantage of things or *make* use of something, rather than 'make advantage' or 'take use'. The use of one word over another depends not only on meaning but also on the surrounding context.

For this reason, the program created for this thesis will be designed to work with phrases. It will assume no prior knowledge of what the phrases are, but based on their frequency of occurrence in the corpus being examined, the program will extract them and summarize the results for the user. This will be a major step towards making corpora more user-friendly for non-linguists and also making them useful for language learners who may not be proficient in the language they are investigating.

Although most of the algorithms used by this program to search through corpora exist in other programs such as WordSmith, no program has combined these algorithms in order to automate the task of finding phrases within the mass of data that simple searches return. To do this, a simple yet effective algorithm is used in which the results from one search through the corpus are then searched a second and third time to find phrases of multiple word lengths, as opposed to just two-word expressions. As explained in more detail in Section 4.3.2, this gives the program the ability to know that, for example, 'behalf of' is not in itself an expression, but rather a part of the larger phrase 'on behalf of.'

Screen shots of the program are available in Appendix A. When the program is first started, the user is shown a brief tutorial explaining what the program does, what a corpus is, which corpora are available, and other useful information. This tutorial is also available from the Help menu. Figure A.1 shows the Main Form[1] of the program with each of the components labeled. The names of the components will be used frequently throughout this thesis, particular in Chapter 4 'Creating the Program.' Figure A.2 shows an example of a search for the word 'hold' in the Brown and Frown corpora. The user can select a corpus by clicking on the File Button or by selecting File → Load Text... from the main menu. This opens the Directory dialog box shown in Figure A.3. If more than one text is selected, the user will be prompted to provide a name for the corpus. This way the selected texts will be treated together as single corpus, and the user can select the corpus again by using

[1]For consistency I have used capital letters in the names of all forms, classes and visual components of the program

the Select Text drop-down box on the main form. When the program starts, any previously selected corpora will be loaded into the drop-down box. Users can use the drop-down box to switch between corpora at any time, allowing a very easy means of comparing the results of a single search in more than one corpus.

I have called the program Phraser (at least until I can think of a better name). The user can enter words via the Enter Word edit box on the Main Form. Phraser then displays a list of common expressions containing that word, along with their frequency in the corpus. The user can then click any expression to see examples of how it is used. The expressions are listed in the scrollable List Box in the top right corner of the Main Form. The last item in the List Box is always 'View all occurrences of. . . ', followed by the word the user entered. If no expressions are found, this will be the only item in the List Box. Example sentences containing the expression are shown in a large memo component called Display that takes up most of the Main Form. It is like the blank area of a word processor, but only used to display text. The Display uses rich text format (RTF), rather than plain ASCII text, so that the expressions can be highlighted in each sentence.

I decided to show the expressions in the context of complete sentences rather than the key-word-in-context (KWIC) concordance used by other programs because I have found that it is much easier to see how words or expressions are being used with complete sentences. The KWIC format is useful for noticing collocates of the keyword, but since this program automatically does this for the user, there is no need to show the results in a KWIC concordance.

Sometimes however, a single sentence is not enough to fully understand how an expression is being used. For this reason, users can double click on any sentence to see more of the context. This brings up a separate form called Viewer showing the sentence that the user clicked on (highlighted) along with 2400 bytes of surrounding text, which is enough to fill up the screen. Figure A.4 in Appendix A shows an example from the phrase 'hold on.'

1.1 Thesis organization

Chapter Two of this thesis talks in more detail about some of the previous attempts to use corpora to help second language learners. It is not intended as an exhaustive account of every work done in this area, as the pedagogical use of corpora is a very large field, and much of it has little to do with this thesis. It does, however, discuss some of the major studies done in this area. It also talks about some of the other programs used to search corpora.

Chapter Three talks about the corpora used in this thesis to test the program. There are four in all[2]: The Brown and Frown corpora, the Switchboard corpus, a Wall Street Journal corpus, and an academic corpus.

Chapter Four shows how the program was created. Flow charts of various functions and event handlers are available in appendix B, while the programming code itself is available in appendix C.

Chapter Five is devoted to using the program. Four example searches are given and their results are discussed. In addition, the program was tested using two ESL students: one beginner and one advanced student. Chapter Five also discusses some improvements and further developments that could be made.

[2]Actually there are five, though Brown and Frown are used together as a single corpus.

Chapter 2

Review of literature and other programs used with corpora

One of the most important early projects that used corpus data to help second language learners was the construction of the Cobuild dictionary, published in 1987. The dictionary was based on a 20 million-word corpus (200 million for the second edition, published in 1995), and its most distinctive feature was the use of clear, plain-language definitions, using complete sentences and, when possible, using only words that are more common than the word being defined. A typical definition is the one for 'motive': "Your motive for doing something is your reason for doing it"(Sinclair 1995). The second edition even gives an indication of how frequently each word occurs in the corpus so that learners (or teachers) can decide how important it is for their vocabulary.

Cobuild is a big project, and the corpus used in creating it, called the Bank of English, is now one of the largest corpora in existence. Most studies dealing with the pedagogical use of corpora are much more small-scale, however, often confined to a single classroom situation. Until recently, the most common approach to using corpora in second language learning has been in the creation of teaching materials for the students. Wooldridge (1991) for example made KWIC concordances from Georges Simenon's *Le Chien Jaune* to find sentences that illustrate the use of certain groups of vocabulary items, such as words related to poison, as well as features of French vocabulary that are often difficult for speakers of English, like the difference between *savoir* and *connaître* 'to know.' The students themselves do not sit down at the computer with access to the corpus, which in this case is just a single novel. From the students' point of view, then, this experience may not be much different than other types of activities they do in class. Although using a corpus in this way may be valuable for creating better teaching materials and finding real-word examples for particular grammatical or lexical features, the students are not necessarily aware that anything has changed in the way they learn.

A different but related use of corpora in second language learning is the creation and analysis of so-called learner corpora, that is, corpora of language produced by people learning English. Again, this is mainly used to help teachers and designers of teaching materials, as well as linguists studying second language acquisition. By systematically analyzing the language of non-native speakers learning English, for example, researchers can refine their approach to EFL error analysis. Sylviane Granger (1998) notes that with learner corpora, researchers can investigate the overuse and/or underuse of certain features of the target language, where as before most work focused only on errors. Geoffrey Leech (1998) lists several other uses of learner corpora, including comparison of linguistic performance by speakers of different linguistic backgrounds (i.e. native language transfer), and helping learners make use of the full range of expressive possibilities in a language.

The analysis of learner corpora is certainly an important field, especially when the results are compared to other corpora. A corpus of native-speaker language, like the ones used in this thesis, provides valuable information on what is common and normal in a language, but it does not give any information about the kinds of words or grammatical constructions that are difficult for non-native speakers. However, this type of research is also distinctly different from the use of corpora that I am proposing in this thesis. The use of learner corpora has a lot in common with the use of native-speaker corpora by teachers and designers of teaching materials in that both approaches add a new tool to a traditional style of teaching. Neither approach puts the learner in charge of analyzing the data, an approach that is quite different from any previous method of language learning.

Johns (1991) was one of the earliest studies arguing for a radically different method of learning known as data-driven learning. In traditional methods, Johns argues, the teacher's role is to impart his or her knowledge of the rules of the target language, then evaluate the students to make sure that they have successfully mastered these rules. This approach assumes that the teacher is already aware of all of the grammatical rules of the language, something which linguists know cannot be true. In data-driven learning however, the teacher guides the students in a process of discovery, using corpus data to answer the students' questions about the language. In this approach, both the teacher and the students learn together. Johns gives an example in which one of his students provided an explanation for when to use 'convince' vs. 'persuade' that turned out to be more useful for the other students than his own explanation.

Data-driven learning was made possible by the rapid advances in computer speed and memory that began in the early nineteen nineties. Still, when compared to today, the resources available for most of the early studies were quite limited. For this reason, most early attempts at data-driven learning focused on smaller corpora, sometimes consisting of just one or two texts. Mparutsa et al. (1991) for example, used very small corpora, ranging from 6,000 to about 30,000 words, to help students at the University of Zimbabwe, where all instruction is done in English. The case studies focus on academic English, noting that most of the students come to the university having learned English from literary sources and often find it difficult to understand the language used in textbooks. Through concordances, they note that many words have a more narrow set of meanings when used in a particular academic field, as with 'demand' and 'equilibrium' in economics.

Hunston (2002) summarizes some of the benefits of data-driven learning in general. Since students are encouraged to use corpus data to answer their own questions, they are frequently more motivated to learn about the language and usually remember what they have learned better than students who learn via grammar books. In has also been hypothesized that data-driven learning improves students' ability to learn from context, something which becomes increasingly important as students move away from controlled classroom situations to real world situation in which they must communicate the target language.

Bernardini (2000) was one of the first ‘classroom concordancing’ studies to make use of large corpora, rather than the small, usually specialized corpora used in previous studies. The study is based on a seminar in which Italian students of English translation used an online version of the British National Corpus to solve a particular research question. Bernardini seems to advocate putting language learners in the role of corpus linguists. In the seminar, students were asked to come up with their own research question and use the corpus to answer it. In a more recent study, Bernardini (2002) attempts to find ways of holding the students’ interest in the corpus activity, since she noticed that the initial enthusiasm in some students working with corpora eventually diminished. She suggests the use of more than one corpus as well as a variety of tools (i.e. software) for analyzing them. In both of these studies, the students seemed to enjoy the activity and could see its usefulness for learning English. But it also made them aware of the fact that learning a language is a life-long activity, due to the subtleties of the language that they wound up investigating in their projects. This may be a good thing in that it encourages them to move beyond oversimplified grammar book explanations, but at the same time it can also be quite intimidating, giving them the false impression that they need to know every detail of the language to be able to use it proficiently.

Along these same lines, Mair (2002) focuses on the use of corpora to help non-native speakers gain native or native-like competence in English. He uses the example of the relatively rare construction ‘possibility to do something’ as opposed to the more typical ‘possibility of doing something’, of which the former occurs 29 times in the British National Corpus, a 100-million-word corpus.

Few studies of this type question the value of the software that is being used to make concordances from the corpus. One exception is Wible et al. (2002), who create a ‘lexical difficulty filter’ for the concordance. One major problem of using concordances in the classroom is that there is no control over the level of the difficulty of the sentences taken from the corpus. The filter they created thus ranks the sentences in order of difficulty by flagging low-frequency words in the context of the keyword. The result is that the sentences near the top of the list tend to be the best sentences for illustrating the use of the keyword.

Most of the software used in these studies have been fairly simple concordancing programs. None of the studies talk in depth about the software used, nor do they usually justify its use over another program. One of the most common programs used in recent years is WordSmith, and in terms of the number of features that it offers, it is one of the most useful for corpus researchers. In fact, many of its features provided the inspiration for my program. Its creator, Mike Scott, provides a useful summary of many of its other features in Scott (2001).

WordSmith creates word lists and concordances as well as any program. It also calculates collocates, a useful feature that is discussed further in Chapter 4. It can show the spatial distribution of the keyword in the corpus and can statistically compare more than one corpus, just to name a few of its features. Whether or not any of the additional features would be useful for second language learners is another question, since the program was designed for linguistic analysis, not language learning. The answer is most likely that they are not, or at least that they would require substantial training before they could be useful.

Unlike most early concordance programs, WordSmith includes a Windows-style GUI (graphical user interface), a feature that increases its ease of use among students who are familiar with using Windows programs. Despite the user interface, it is not always straightforward to use the program due to the large number of buttons and menus the user must click through to make a concordance. Unlike my program, it does not put its most important features in a single window. One way to evaluate the ease of use of a Windows program is to count the minimum number of clicks on either the mouse or the keyboard that are required to perform a particular task. Using WordSmith version 4, it requires a minimum of twelve clicks to create a concordance starting from the program's opening screen. This count does not include the entry of the keyword itself. With my program the minimum number of clicks is one, the clicking of the OK Button. To be fair, seven of the clicks in WordSmith were for selecting a corpus. If the user wants to use a corpus other than the default corpus on my program, an additional two clicks are required if the corpus has been previously loaded, and four clicks if it has not. Either way, this shows that in terms of its ease of use, WordSmith has a lot of room for improvement.

Another useful program is the Cobuild Concordance, a demonstration of which is available for free on the internet (http://titania.cobuild.collins.co.uk). The demonstration searches the Bank of English, the same corpus used to make the second edition of the Cobuild dictionary (Sinclair 1995). One of the nice features of this program is that it takes advantage of part-of-speech tagging, allowing users to search for keywords in a specified context. For example, if a user was interested in phrasal verbs, he or she could enter a verb and specify that it must be followed by a preposition. The code for a preposition in this program is 'IN', so a search for 'put IN' would return a concordance of 'put' followed immediately by a preposition. A plus sign plus a number indicates the span of words to be considered. For example, 'put+3IN' would find the word 'put' followed by a preposition within a span of three words to the right, as in '**put** an end **to** these things.'

This is a very useful tool for finding examples of specific types of phrases. It does, however, require users to know in advance what type of phrase they are looking for. There are eighteen codes for different parts of speech, which fortunately is not overwhelming in comparison to the level of detail most part-of-speech taggers include, but it is enough so that it may never occur to many learners to try certain combinations. The Cobuild Concordance thus falls into the same category as WordSmith: a very powerful tool, but ultimately better suited to people who already know a lot about English and particularly to linguists.

At this stage in its development my program does not offer as many features as WordSmith or the Cobuild Concordance. Its purpose for now is to illustrate that second language learners do not have to become experts in corpus linguistics, as studies by Johns (1991), Bernardini (2000, 2002) and others imply. Like the 'lexical difficulty filter' described in Wible et al. (2002), my program attempts to find in the corpus what is useful for the learner, rather than overwhelming the user with all of the subtleties and minute details of the language.

Chapter 3

Description of the corpora used

Part of what makes this program a useful tool is the ability to use it with any corpus. The types of phrases that the program extracts depend entirely on which phrases are used in the corpus. This is beneficial for students wishing to become more proficient in a particular style of English. For example, a foreign student entering into an American university will want to use an academic corpus, consisting perhaps of excerpts from textbooks, transcripts of lectures, and other types of language that he or she is likely to encounter. On the other hand, a person entering into the business world would want to choose a business-oriented corpus. If this program is marketed in the future, it could even be packaged with a variety of corpora that the user could select.

At this stage of the project, however, I have selected corpora mainly just to be able to demonstrate how the program works and show how it is useful. Due to limited resources I have not been able to obtain the ideal corpora for the job. For an academic corpus, an ideal choice, at least for the spoken portion of it, would be the Michigan Corpus of Academic Spoken English (MICASE). It consists of transcripts of academic speech collected from the University of Michigan, including lectures, student-teacher meetings, and study groups. But since this corpus was not available, I created a corpus of academic writings freely available on the internet. For a business corpus, I used a Wall Street Journal corpus. The corpus was designed to train part-of-speech taggers, so unfortunately no information is available on its exact contents.

Since many users of Phraser may not want to limit the texts they search to a particular style of English, I have also used the Brown and Freiburg-Brown corpora together as a corpus

Table 3.1: Sizes of corpora used

Corpus	**Total Words**
Brown and Frown (combined)	2,017,864
Switchboard	1,376,306
Academic	957,212
Wall Street Journal	958,420

of general written American English. These corpora include samples of writing from many different genres, including both fiction and non-fiction. They do not, however, include any samples of spoken language. For this, I have used the Switchboard Corpus of transcribed telephone conversations.

All of the corpora used in this thesis are roughly between one and two million words. Table 3.1 gives the exact word counts, as obtained from WordSmith. The complete version of the Switchboard corpus contains approximately three million words, though only about half of the corpus has been used in this program. Part of the reason for this is that anything larger than two million words becomes much more difficult to search electronically. From experimenting, I have found that a corpus of one to two million words is sufficient for finding commonly used phrases. However, I have also realized that more is always better since many expressions do not occur within one to two million words of text. One could argue that for second language learners, any expression that does not occur frequently within two million words is not likely to be an important one. This, however, entirely depends upon the students' level of proficiency in English. For beginning and intermediate students, a one to two million-word corpus is probably sufficient. However, to be used by more advanced language learners wishing to fine-tune the details of the language, or to be used as a reference guide, a much larger corpus would be needed.

The algorithm currently used in Phraser to retrieve words from the corpora is fairly simple; to work with larger texts of more that two million words, a more sophisticated algorithm will be needed. This thesis is not intended to explore the intricacies of text retrieval, however, so this part of the program is not yet fully developed. With the exception of the Switchboard corpus, none of the texts were modified in any way. The texts that form the academic corpus were saved as plain (ASCII) text, so some formatting in the original texts may have been lost. The Switchboard corpus was converted from XML to plain text in order to increase the readability of the sentences from the corpus.

The rest of this chapter describes these corpora in more detail, though unfortunately no information is availble on the contents of the Wall Street Journal corpus.

3.1 Switchboard Corpus

For a spoken corpus, I used the Switchboard Corpus of telephone conversations. It is not the ideal spoken corpus to use, since the speakers had to choose from a set of fifty-two topics for the conversations. The topics include things like clothing, political issues, vacation, hobbies, sports, and family life. One of the disadvantages is that certain words related to these topics are over-represented in the corpus, while other words related to other topics may be under-represented. Before calling, each speaker selected several topics that they would feel comfortable discussing. Most speakers participated more than once, speaking with a different participant each time about a topic that both speakers had listed as a possibility.

The corpus was designed for speech recognition research, so it was never meant to be used as a representative sample of spoken American English. It does, however, inlcude data from over 500 speakers selected in an attempt to represent all of the major dialects of American English, though it only includes speakers whose native language is English. The transcripts, which were done by court-recorders, are very readable and do not attempt to represent the speakers' pronunciation by using various spellings for the same word (such as 'looking', 'lookin' ', etc.)

3.2 Brown and Frown Corpora

The Brown and the Freiburg-Brown (commonly known as Frown) corpora were used together as a general corpus of American English. The Brown Corpus consists of approximately one million words of written American English published in 1961. The Frown Corpus is a re-creation of the Brown corpus, using the same text-types, but from writings published in 1991. Although Brown was intended to be a balanced corpus, it, along with Frown, contains a disproportionately large number of texts from certain types of writing, especially journalistic writing. They are divided into informative writings (press reportage, reviews, academic papers, etc.) and imaginative writings, including various genres of fiction. They do not include any samples of spoken language, and they intentionally avoid dialog in the fiction samples. But as general corpora of written American English, they are certainly useful.

The Brown Corpus contains 500 samples of writings, each containing at least 2000 words. Table 3.2 summarizes the types of texts that it includes, along with the number of samples of each type. The Frown corpus used these same categories and sample sizes in order to provide an updated corpus of comparable content.

3.3 Academic Corpus

The academic corpus is the only corpus that I created for this project. It is composed of twelve academic books from various fields, such as biology, economics, and psychology. The styles of the texts range from very formal, textbook-style writing like the molecular biology textbook to the more colloquial, almost conversation style of Neil Stephenson's *In the Beginning... There Was the Command Line*. The texts were gathered from various websites and were all freely available. The only criteria used in selecting them were that a citation for

Table 3.2: Contents of the Brown Corpus

Text Type	**Samples**
Informative Prose (i.e. non-fiction)	**374**
Press: Reportage (eg. news, politics, sports)	44
Press: Editorial	27
Press: Reviews (eg. theatre, books, music)	17
Religion (books, periodicals)	17
Skills and Hobbies (mostly periodicals)	36
Popular Lore (books, periodicals)	48
Belles Lettres, Biography, Memoirs, etc. (books, periodicals)	75
Miscellaneous (mostly government documents)	30
Learned (i.e. academic writing)	80
Imaginative Prose (i.e. fiction) (mostly novels and short stories)	**126**
General Fiction	29
Mystery and Detective Fiction	24
Science Fiction	6
Adventure and Western Fiction	29
Romance and Love Story	29
Humor	9

the text must be available elsewhere (to ensure that it is a real, published academic piece of work) and it must have been published in the last fifty years. The texts included are listed in Table 3.3.

This collection of texts is by no means meant to be a representative corpus of academic writing. Although I have made an effort to include texts from a broad range of academic fields, the choice of which texts to include was largely based on what was available. This is not a problem at this stage of the research, however, since the purpose of the corpus is to illustrate the use of the program. A much more carefully designed corpus will be necessary if the program is marketed in the future.

Table 3.3: Contents of the academic corpus

Title	**Genre**	**Size**
Bits of Power: Issues in Global Access to Scientific Data	Science/Communica-tion	328 KB
Do Teachers Care About Truth? Epistemological Issues for Education	Education	182 KB
Earthdance: Living Systems in Evolution	Ecology	793 KB
Economic Policy: Thoughts for Today and Tomorrow	Economics	181 KB
Ethnocriticism: Ethnography, History, Literature	Anthropology	371 KB
From c-Numbers to q-Numbers [Parts A and B]	Physics/Mathematics	692 KB
The Evolving Mind	Philosophy	414 KB
In the Beginning was the Command Line	Computer Science	208 KB
Limits of Liberty: Between Anarchy and Leviathan	Political Science	509 KB
Molecular Cell Biology [Chapters 1 - 8]	Biology/Chemistry	773 KB
The Nuclear Energy Option: An Alternative for the 90s	Nuclear Industry/ Government Policy	825 KB
The Romance of American Psychology: Political Culture in the Age of Experts	Psychology	756 KB

Chapter 4

Creating the program

The program is written in C++ using the C++ Builder Integrated Development Environment (IDE). It is a stand-alone Windows application with a typical graphical user interface (GUI). I chose to make the program as a Windows application instead of a web-based application for two reasons. The first is that I am already familiar with making programs using C++ Builder, and I have no experience in making web applications. The other reason is that as a future project, I would like to test the usefulness of the program in Brazil with students learning English there. A web-based application requires that the user have internet access, which is not as readily available in Brazil as in the United States. A stand-alone application on the other hand can be distributed on CD-ROM or diskette; thus anybody with access to a computer can use it.

All of the corpora used have been indexed so that the program can quickly retrieve words from them. Working with large amounts of texts electronically is not a simple matter. An algorithm that requires a character by character search through the corpus each time the user enters a word would be inefficient and unbearably slow. Fortunately, as Witten et al. describe in *Managing Gigabytes* (1999), creating an index allows the computer to locate any word in the text with remarkable speed. However, many of the algorithms described by Witten et al. were designed to work with several gigabytes of text that contain nearly a billion words. These algorithms tend to be very complex, and developing such a text retrieval system would be another large project in itself. For this reason, I have used a very simple indexing system, which works quite well for up to 10 megabytes of text, or approximately two million words. The index used is described in more detail in Section 4.2.1.

4.1 Classes and events

C++ Builder automatically creates a class for each form in the program.[1] The programmer is free to create other subclasses within each form class, but they will all be descendents of a form class. However, I usually find that the classes created by C++ Builder are sufficient since each form roughly corresponds to a single task that the program performs. Therefore I have not created any classes of my own for this program. Screen shots of all of the forms in this program are shown in Appendix A. The three major forms have already been mentioned: the Main Form, the Directory, and the Viewer. The following summarizes all of the forms and classes and what they are used for. Form names are always the same as the class name.[2]

MainForm (Figures A.1 and A.2) The main window of the program, in which the user can enter words and view phrases and sentences. Most of the program is contained within the MainForm class.

Directory (Figure A.3) The dialog box that allows the user to select a text or corpus to be searched. Its most important event is the OKButtonClick, described in Section 4.6.

Viewer (Figure A.4) A text viewer that is displayed when the user double clicks on a sentence. Shows the context in which the sentence occurs in the corpus.

NameCorpusForm (Figure A.5) Prompts the user to provide a name for the corpus if he or she selects more than one text.

ProgressForm (Figure A.6) Displays a progress bar; shown when a text is loaded and while searching for phrases.

StopWordsForm (Figure A.7) Allows the user to select which types of stop words should be used when searching (discussed in Section 4.3.5).

[1]For Windows users, a form is simply a window that can be opened and closed. A class in programming is a set of variables and functions that work together to perform a particular task.

[2]In C++ these names are written without spaces as I have done in the list.

StopWordsMsg (Figure A.8) Displayed before showing the StopWordsForm; explains what stop words are used for.

Like most Windows programs, Phraser is event driven, meaning that instead of performing a single task, it responds to events initiated by the user, such as entering text or clicking on a button. C++ Builder automatically generates all of the code used to create the visual components of the program, including menu items, buttons, and the form itself. All the programmer has to do to create the user interface is place the components on the form and set their properties as desired. Most components also have event handlers in which the programmer can write the code that is run when the user does something, such as clicking on a button.

Event handlers are simply functions that belong to a particular form (i.e. class). In C++, the type name of the class is written first (Borland C++ always prefixes type names with a T), followed by two colons and the function name (i.e. the name of the event). The five major events in this program are:

TMainForm::FileLoadClick Occurs when the user clicks on the File Button or selects File → Load Text… from the main menu. This opens the Directory form and waits for the user to close it. If the user clicks the OK Button, it loads the index of the text or corpus into memory so that it can be searched. If no index is available, it creates one.

TMainForm::OKButtonClick Occurs when the user clicks the OK Button on the Main Form. If there is a word in the Enter Word edit box (under the 'Enter a word' label), the program finds that word in the corpus and returns phrases containing the word.

TMainForm::ListBoxClick Occurs when the user clicks on the List Box that contains the list of phrases. All sentences containing the phrase the user selects are displayed in the Display component.

TMainForm::DisplayMouseUp Occurs when the user clicks on the Display, the rich text component that displays the sentences containing the selected phrase. This is used to open the Viewer form that displays the larger context in which the sentence occurs.

TDirectory::OKButtonClick Occurs when the user clicks the OK Button on the Directory form. This lets the user select a text or texts in which to search for phrases. If the user selects more than one text, it creates a file that links the texts together.

In addition, there are event handlers that are called immediately after starting the program and immediately before closing. The rest of this chapter describes these events in more detail, showing the algorithms used to perform each task.

4.2 TMainForm::FileLoadClick

Clicking on the File Button or selecting File → Load Text... from the main menu opens the Directory dialog box as show in Figure A.3 in Appendix A. This allows users to select one or more texts to use as a corpus. The event handler calls the LoadText function, passing the name of the corpus as a parameter. A flow chart representation of this function is in Appendix B, Figure B.1.

The LoadText function does one of two things. If an index is available for the corpus the user selected, it loads the index into memory in the form of three vectors. If no index is found, it creates one and saves it to disk so that it will only have to be created once. Creating an index takes several minutes; though it depends on the size of the corpus and the speed of the computer, it typically takes ten to twenty minutes for the corpora used in this thesis. Once the index is created, however, it only takes a few seconds to load it into memory, so users do not have to wait that long unless they have selected their own texts that have not been indexed.

Indexing allows for words in the corpus to be located very quickly. It is analogous to the index in the back of a book and, in this case, actually works very similarly. The major difference is that the index in a book only includes certain key words that may be of interest to the reader, while the index for this program includes almost every word in the corpus. Thus the index is nearly the size of the corpus itself. In order to reduce the size of the index as well as reduce the amount of time it takes to create it, extremely high-frequency words such as articles, conjunctions, and common prepositions are not included in the index. This greatly reduces the size of the index since these high-frequency words make up a very large percentage of the actual word tokens in any English text.

One result of this, however, is that the user will not be able to search for any of these words. This is not a problem for this program for two reasons. First, because of the way the search algorithm works, searching for phrases with these words would take several minutes, regardless of how fast the words are located in the corpus. But more importantly, even if the user had the patience to wait for such a search, the results are not likely to be valuable for a second language learner. The resulting phrase list would be so large that the amount of data would be overwhelming. A search for an article for example would essentially return a list of the most frequent nouns in the corpus, such as 'the time', 'a year', and 'an order'. For this reason, if the user enters one of the words excluded from the index, a message is displayed explaining why it is not a good idea to search for that word, along with some suggestions of other types of words that might provide a more useful search. (See Section 4.3.6 for more details on these messages).

The other major difference between a file index and the index in a book has to do with what the numbers in the index refer to. In a book, they generally refer to page numbers. In most file indexing systems they refer to the byte offsets in the file. In this program, however, they refer to the sentence in which the word occurs. Thus '0' indicates the first sentence, '1' indicates the second sentence, and so on. A separate file is used to store the byte offsets of each sentence. The advantage of this method is that the index actually serves two purposes: it is used not only to locate words in the text, but also to parse the text into sentences.

4.2.1 Creating the index

If no index is available for the file or files that the user selects, the program will create one. Since this will take several minutes for even a moderately sized corpus, the user is prompted with a message before continuing. A progress bar keeps track of how much time is remaining. In this case, the progress bar is very accurate because the time it takes to create the index is directly related to the size of the corpus. Thus the maximum value of the progress bar is set as the number of bytes in the file, and it is increased by one for each byte processed. The number displayed to the user is the percentage of completion.

The corpus is opened in binary mode so that each byte will correspond exactly to one character, which may be a letter, a symbol, a space, a new-line character, or a carriage return. If the corpus is tagged, the tags (anything enclosed in <>) are for the most part ignored. Many corpora include tags that mark the beginning and end of each sentence, so if this is true they are used to determine where the sentences begin and end. If not, the program uses the punctuation markers '.', '!', and '?' to mark the end of a sentence, though this will inevitably lead to some errors if the corpus also uses periods in abbreviations as well. (Some, like the Brown corpus, use a period to mark the end of a sentence and a different symbol elsewhere.) The byte in the file at which each sentence ends is stored in a vector and eventually saved to a file with the extension '.sen'. These numbers thus serve to parse the text into sentences, since every sentence will begin at the byte corresponding to one number and end at the next.

The index itself is actually two files. One, which is given the extension '.wdx' (for '**w**or**d** inde**x**'), is simply a word list containing every word in the corpus in alphabetical order. The other, which is parallel line by line to the word list, contains the numbers corresponding to the sentences in which each word occurs. This file is given the extension '.vlx' (for '**v**a**l**ue inde**x**'). To locate a particular word in the corpus, the word is quickly located in the word index using a binary search, then the corresponding values for that word are obtained from the value index. The numbers in the value index can then be used with the list of sentence byte offsets to extract from the corpus each sentence in which the word occurs. This is explained in more detail in Section 4.3.1.

Once the index is created, it is saved in the form of three files as described above, and it is retained in memory so that it can be used during subsequent searches. If the index already exists on file when the user loads a particular text, the index is simply loaded in the memory in the form of three vectors, one for each of the files. The IndexWords vector corresponds to the word list ('.wdx') file, IndexValues corresponds to the value index ('.vlx') file, and Sentences corresponds to the file containing byte offsets of the sentences ('.sen'). Loading the index into these vectors is much faster than creating an index, though it does take a few seconds. For this reason a progress bar is displayed saying 'Loading text…'. In this case, it is impossible to know in advance exactly how long it will take to load the index, since it requires three loops to load it (one for each file/vector). Thus the time is at first estimated, then adjusted after the first loop.

If the corpus consists of more than one text, a fourth file is created with the extension '.cor' which serves to link the files together.[3] It is a relatively small file containing the file name of each text along with its size in bytes. The index in this case is created for all of the files together, and the '.cor' file is used to determine which file a particular sentence is in. Two vectors store this information when the user loads a particular file: Files contains the names of each of the files in the corpus, and FileSizes contains the size of each file. If the user selects only one text, these vectors will only contain one element, the name of that file and its size.

All five of these vectors—IndexWords, IndexValues, Sentences, Files, and FileSizes—are variables that belong to the MainForm class and are available to any function in the class.

[3]Actually this file is created in the TDirectory::OKButtonClick event handler, when the user selects the texts. See Section 4.6.

In particular they are used by the Concord (Section 4.3.1) and GetParagraph (Section 4.5.1) functions.

4.3 TMainForm::OKButtonClick

The OKButtonClick event is the main part of the program. This event searches the corpus for phrases containing the keyword (the word the user entered) and displays them, along with the sentences in which they are found. Each time the event is called, four variables are created (after destroying any previous ones): two StringLists and two vectors. A StringList is a class provided by C++ Builder to store and manipulate lists of strings. Vectors are part of the C++ Standard Template Library and can be used to do many of the same things as StringLists. They are not identical however and are used for slightly difference purposes. Specifically, StringLists are more closely tied to the visual components that C++ Builder provides, so I have tended to use StringLists for lists that will eventually be displayed to the user and vectors for everything else.

The StringLists created are called Concordance and PhrasesList. Concordance contains all of the sentences from the corpus that contain the keyword. PhrasesList is the list of phrases containing the keyword, including the frequency of each phrase in the corpus. The two vectors created are called Phrases and PhrasesCount. Phrases contains just the phrases and PhrasesCount contains their frequency in the corpus. The items in these vectors are eventually concatenated and added to PhrasesList.

After creating the variables to hold the phrases, the next step is obviously to find them. As this is a multi-step process, several functions are used to do this. They are, in the order in which they are called:

Concord Using the index, this function locates all the sentences that contains the keyword and stores them in the Concordance StringList. Concordance is a class variable available to all functions in the class.

GetPhrases This function finds phrases containing the keyword, using a multiple search algorithm as described in Section 4.3.2. It calls the function FormPhrases to actually find the phrases.

FormPhrases To locate phrases, this function does two things. First it creates a list of collocates one word to the left and one word to the right of the keyword. Then it joins the collocates with the keyword to form phrases.

IsStopWord This is a function called by FormPhrases. It determines if a given word is a stop word, as described in Section 4.3.5

DisplayMessage Displays a message if the user enters a high-frequency word that was excluded from the file index.

SortLists This function sorts the phrases by their frequency. Since the phrases are stored in two vectors, one containing the phrases and the other containing their frequency, the sorting must be done manually so that the contents of one vector will correspond to the contents of the other.

Other functions are used to perform other minor tasks, but these are the main ones. Most of them do not return a value; instead, they modify the variables that were created at the start of the OKButton event handler, adding values to the StringLists and vectors as necessary. The only functions that return values are Concord and FormPhrases. Concord returns the frequency in the corpus of the keyword. FormPhrases returns the number of phrases found during a given search.

A flow chart for this event handler is in Appendix B, Figure B.2. If the keyword is not found, a message is displayed to the user saying so, and the event handler does not continue after calling Concord. If the keyword is one of the words excluded from the file index, a message is displayed saying why it is not a good idea to search for such a word, along with suggestions for other types of words that might be more productive (see Section 4.3.6 for more information on these messages). Otherwise the event handler proceeds to GetPhrases. If no phrases are found, the user is prompted with a message as well, and all of the sentences containing the search word are shown. This way, even if the program does not find any expressions, the user is at least shown something that may be useful. If phrases are found, they are sorted using SortLists, then displayed in the ListBox on the MainForm. To display the sentences, a function called DisplayPhrases is called. This function is responsible for highlighting the phrase in each sentence, which it does by setting the text of the phrase to bold and changing the color to blue. This function is described in more detail in the TMainForm::ListBoxClick event in Section 4.4.

The rest of this section is devoted to describing the functions mentioned above in more detail. Flow charts for the following key functions are available in Appendix B: Concord (Figure B.3), GetPhrases (Figure B.4), and FormPhrases (Figures B.5 and B.6). The C++ code itself is in Appendix C.

4.3.1 Concord

The Concord function creates a concordance of the word that the user entered (the keyword). It uses the index described in Section 4.2.1 to locate the word in the corpus and uses the list of byte offsets of the sentences that was also created as part of the index to load each sentence containing the word into a StringList called Concordance. This StringList is kept in memory until the user clicks the OK Button again, at which time it is deleted and a new one is created. (This is done outside of the function; the function simply fills Concordance with data.)

As described in Section 4.2.1, the index consists of a word list and a corresponding list of values for each word, with each value referring to a sentence number. Both the word list and the value list are stored in vectors called IndexWords and IndexValues by the time Concord is called. The first step is to do a binary search on the word list (IndexWords) to locate the keyword. Then, the corresponding entry in IndexValues is obtained.

The values for each word are separated by commas, so these are parsed into a another vector called index_vals. Since they are saved to disk as strings, the values are also converted to integers at this time.

Next, Concord uses the three other vectors that were created in the MainForm::FileLoadClick event described above in Section 4.2. The values in index_vals refer to the sentences that contain the keyword, so for each of these values, the byte offsets of the sentences are obtained from the Sentences vector. However, since the corpus may consist of several files, the Files and FileSizes vectors are used to determine which file each sentence is in. Once the byte offsets and the file are known, the sentence can be read from file and stored in the Concordance StringList.

The Concord function returns the size of index_vals, which is the number of occurrences of the keyword. This number is used as the last entry in the ListBox, which allows the user to view all occurrences of the keyword.

4.3.2 The Search Algorithm

Once Concord has created a concordance of the keyword, containing each sentence from the corpus in which the word occurs, the next step is to extract phrases. This is done in two functions, GetPhrases and FormPhrases. The reason for having two functions will become clear after I explain the multiple search algorithm.

The algorithm that Phraser uses to find expressions involves counting the number of collocates of the search word, then forming Phrases based on these collocates. This is similar to the way WordSmith's "Collocates" program works. In this program, WordSmith counts the number of occurrences of words to the right and to the left of the keyword. The most frequently occurring words are displayed to the user in the form of a table showing how many times each word occurs in a span of five words to the left and five words to the right of the keyword. Table 4.1 shows an example with the word 'home', sorted by the collocates one word to the left (L1). This example shows that, in the Brown and Frown corpora, the word

Table 4.1: Collocates of the word ‘home’ in the Brown and Frown corpora (adapted from WordSmith).

WORD	L5	L4	L3	L2	L1	CENTRE	R1	R2	R3	R4	R5
AT	12	6	11	25	139		17	3	5	2	5
THE	57	71	45	54	64		40	74	57	58	73
HIS	7	2	10	16	44		3	7	4	9	9
A	22	22	20	25	34		11	29	34	25	28
GO			1	3	30			1		1	1
THEIR	7	3	5	10	28		2	6	5	3	2
FROM	4	3	4	12	23		26	2	3	7	6

‘at’ occurs 139 times immediately to the left of home (e.g. ‘at home’), 25 times two words to the left of home (e.g. ‘at her home’), and so on. A similar type of table is constructed by Phraser; however, none of it is displayed to the user.

In order to use this information to find expressions, the collocate table needs to be modified. Rather than calculating the collocates that occur within a span of five words on both sides of the keyword as WordSmith does, Phraser uses a multiple search algorithm, increasing the span of collocates with each search. The first step is to count the number of collocates that occur immediately to the left and to the right of the keyword, then sort them by frequency. Using the most frequent of these collocates, the program can generate a list of two word phrases, such as ‘at home’, ‘the home’, and ‘home in’. Next, these phrases are used as the keywords in a second search. For example, if ‘the home’ is used as the keyword, the top two collocates in the Brown and Frown corpora are ‘$\sim$ of’ (14 times) and ‘in $\sim$’ (13 times), with $\sim$ representing the location of the node ‘the home’. Thus from the second search, three-word phrases such as ‘the home of’ and ‘in the home’ can be generated. Finally, a third search using the most frequent results of the second search will generate four word expressions. For example, the top collocate of ‘at home’ is ‘$\sim$ and’ (14 times). A search using ‘at home and’ as the keyword would generate ‘at home and abroad’, assuming that ‘abroad’ occurs frequently enough as a collocate (this expression occurs three times in the Brown and Frown corpora). I have decided to stop at three searches rather than continuing with a fourth or fifth. This means that the program will not find expressions longer than four words. However, expressions of that length are not likely to occur frequently in a corpus of one to two million words.

With this algorithm, only the collocates immediately to the left and right of the keyword need to be counted. This does not, however, mean that only the words immediately to the left and right of the word are considered. Because this algorithm performs multiple searches, each search essentially moves out a layer, considering words further away from the original keyword. Since there are three searches, a span of six words is considered (seven including the keyword): three to the left and three to the right. This is generally sufficient to find the most common and useful phrases.

Looking through a list of collocates, it quickly becomes apparent that most of them are high frequency words that do not form expressions. For this reason, it is necessary to use a stop list of words that, if added to the keyword, should not be considered as expressions. A good example of this is the occurrence of articles with nouns. Without stop words, the top expression from a search for any noun is likely to be ‘the $\sim$’.

The composition of the stop list will be discussed further in Section 4.3.5. But for this section the main issue is how to deal with expressions formed from stop words. The term ‘stop list’ normally means a list of words that one does not want included in the results of a corpus search. For this program however, this is not exactly what ‘stop list’ means. It might be tempting simply not to count words included in the stop list at all, thus eliminating them as possible collocates. However, this would be problematic since many expressions do contain articles, conjunctions, and other words from the stop list. For example, for collocates of the word ‘home’ in Brown and Frown, ‘in’ occurs much more frequently two words to the left (29 times) than one word to the left (5 times) of ‘home’. This is because, unlike ‘at’, which is frequently used without an article (‘at home’), ‘in’ is almost always used with ‘the’; thus the expression is not ‘in home’ but ‘in the home’ (example from the Brown corpus: “Woman’s place is in the home”). Such details may seem like common sense to a native speaker of English, but these are precisely the details that are difficult for second language learners. For this reason knowing which words are stop words will only become important after all of the phrases have been formed.

Thus the last phase of the search algorithm will be to eliminate any phrase that was formed by adding a stop word. For example, one of the top collocates of ‘put on’ is ‘$\sim$ a’. But since the article ‘a’ is included in the stop list, the phrase ‘put on a’ should not be included in the final phrase list.

The last thing that needs to be considered in this algorithm is figuring out what exactly is meant by a “frequent collocate”. Obviously some kind of cut-off is needed. The question is essentially “how many times should a phrase occur for it to be considered important?” I have chosen three for this program, although this is by no means a magical number. Since the phrases will eventually be sorted by frequency, I have intentionally kept the number low. If the user enters a common word, like ‘time’, a long list of phrases will be returned. But since they are sorted, the most frequently occurring ones will be at the top of the list. The user will have to scroll down to see less frequent phrases. For this reason, the phrases that only occur three times do not hurt anything; they are there if the user wants to see them, but they do not get in the way. On the other hand, if the user enters a relatively low-frequency word, like ‘repeatedly’, the most frequent phrase might occur only three or four times. Setting the cut-off point high would mean that no phrases would be found. A cut-off of less than three seems impractical though. A cut-off of two dramatically increases the time it takes to find phrases without providing many additional useful expressions, and a cut-off of one would be the equivalent of not having a cut-off.

Using the cut-off might appear straightforward, and in the first search it is. In the second and third searches, however, things get more complicated because it is necessary to consider the new (expanded) phrases in relation to the ones from which they were derived. For example if the user enters the word 'behalf', the top two collocates found will probably be 'on ∼' and '∼ of'. Obviously, 'on behalf' and 'behalf of' are only parts of the complete expression, 'on behalf of'. The way the program deals with this is by keeping track of the total number of longer phrases derived from shorter phrases during the second and third searches. For example, the phrase 'on behalf' occurs 32 times in the Brown and Frown corpora. The top collocate of this phrase is '∼ of', which also occurs 32 times. Thus since $32 - 32 = 0$, there are no additional phrases (i.e. the word 'of' occurs 100% of the time after the phrase 'on behalf'). In such cases, the original phrase 'on behalf' can be deleted in favor of the complete phrase 'on behalf of'. Starting with the other half of the phrase in this example is a little more complicated, but the idea is the same. The phrase 'behalf of' occurs 39 times. The top collocate is 'on ∼', occurring 34 times. The next highest collocate is 'in ∼' occurring 5 times. After subtracting $39 - 34 - 5 = 0$, we see that these two collocates account for 100% of the occurrences of 'behalf of'. Thus this short phrase can be deleted in favor of 'on behalf of' and 'in behalf of'. Even if there were one or two phrases left over after doing the subtraction, it would still be safe to eliminate the shorter phrase. The program uses the same cut-off used above (three) to determine when the shorter phrases can be ignored.

4.3.3 GetPhrases

Since GetPhrases has to perform multiple searches in order to find expressions, it now makes sense why two functions are necessary. GetPhrases is the function that is called first. It in turn calls FormPhrases each time it needs to do a search. To keep track of all the information needed, such as how many phrases were formed from a given search and whether or not a phrase was formed by adding a stop word, several vectors are used.

GetPhrases creates these vectors upon being called and deletes them before returning. There is a vector for each phrase length: TwoWords, ThreeWords, and FourWords. TwoWords contains two-word phrases, ThreeWords contains three-word phrases and so on. Since it is important to know the frequency of each phrase in the corpus, another set of vectors is used to keep track of this: TwoWordsCount, ThreeWordsCount, and FourWordsCount. It is also important to know which phrases were created by adding a stop word, and which side of the keyword the stop word is on. Six boolean vectors are used for this, three for the left side and three for the right side. They are called TwoWordsStopLeft, ThreeWordsStopLeft, FourWordsStopLeft, TwoWordsStopRight, ThreeWordsStopRight, and FourWordsStopRight. If a phrase was formed by adding a stop word, its corresponding value in these vectors is set to 'true', otherwise 'false'. For example, if the search word is 'home' and the phrase formed is 'the home', the corresponding value in TwoWordsStopLeft would be set to 'true', since 'the' is a stop word and it was added to the left side of the node 'home'. The reason it is necessary to know this will become apparent shortly.

After creating these vectors, GetPhrases begins to call FormPhrases to perform the searches. FormPhrases takes several arguments. It is always passed one set of vectors. The first time it is called it is passed TwoWords, TwoWordsCount, TwoWordsStopLeft, and TwoWordsStopRight. Later it is called with the vectors for three- and four-word phrases. FormPhrases also needs to know what the keyword is and, in cases where the keyword is a phrase, whether or not it was formed by adding a stop word and on which side of the keyword it was added. Thus the last three arguments contain this information. For the first search, the keyword is equivalent to the word the user entered and the last two arguments do not apply, so the value 'false' is used.

```
FormPhrases(TwoWords, TwoWordsCount, TwoWordsStopLeft,
    TwoWordsStopRight, Headword, false, false);
```

FormPhrases returns the total number of phrases that are found. For example, if it finds 34 occurrences of 'on behalf of' and 5 occurrences 'in behalf of', it returns the value 39. In the first search shown above, this value is not needed so it is discarded.[4]

[4]The first four arguments of FormPhrases are actually pointers to the four vectors. Thus the function modifies the vectors via the pointers and only the number of phrases found is returned.

Next, all of the phrases found in the first search need to be searched as well. Thus each element of TwoWords is passed to the FormPhrases function.

```
for (int i = 0; i < TwoWords->size(); i++)
   {  int num = FormPhrases(ThreeWords, ThreeWordsCount,
                 ThreeWordsStopLeft, ThreeWordsStopRight,
                 TwoWords->at(i), TwoWordsStopLeft->at(i),
                 TwoWordsStopRight->at(i));
      if (TwoWordsCount->at(i) - num < cut_off)
      {  (*TwoWordsStopLeft)[i] = true;
      }
   }
```

This time, the results of the search are added to the ThreeWords vector. Here, the value that FormPhrases returns (the total number of phrases found) is important, so it is saved as 'num'. If the frequency of the search phrase (in this case TwoWordsCount->at[i]) minus num is less than the cut-off (three), the phrase is marked as having been formed from a stop word. Although this is not actually true—the phrase was not necessarily formed by adding a stop word—it has the effect of eliminating it from the final list of phrases displayed to the user. Thus in the example above of 'on behalf of' and 'in behalf of', the original phrase 'behalf of' would be marked so that it is not be displayed to the user.

Next, the same thing is done for the contents of ThreeWords, with the results stored in FourWords. After all of the searches are completed, a final list is created consisting of all of the phrases from TwoWords, ThreeWords and FourWords that are not marked as having been formed from stop words. This is the list that is eventually displayed to the user.

Although it has been omitted from the code cited above, the GetPhrases function also displays a progress bar to the user.[5] It is impossible to know in advance how long the entire search will take. This depends on how many phrases are found in each of the searches. In general there appears to be a direct relationship between the frequency of the search word

[5]Actually the progress bar has already been displayed by the time GetPhrases is called. The function just increments it.

in the corpus and the number of times that FormPhrases will be called. From experimenting with various words I have found that the frequency of the search word in the corpus is generally between three and a half and four times the number of times that FormPhrases gets called. For example ‘go’ occurs 603 times in the Brown corpus, and during a search for ‘go’, FormPhrases is called 165 times ($603 / 165 \approx 3.65$). Thus the maximum value of the progress bar is set at one-fourth of the frequency of the word in the corpus. This is an overestimation, though with progress bars it is better to overestimate the time rather the leave the user waiting while the progress bar reads ‘100% complete’.

4.3.4 FormPhrases

GetPhrases and FormPhrases work together to find phrases in the corpus. GetPhrases provides the overall structure of the algorithm while most of the messy details are hidden in FormPhrases. FormPhrases gets called once with the word the user entered and several more times for all of the phrases that are found in each search. Thus the keyword in this function may be a single word or a phrase.

The search is performed on Concordance, which is the list of all the sentences containing the word the user entered. To find the keyword, a function provided by Borland called Pos is used. Pos returns the position (starting at 1) in a string of the first occurrence of a substring. If the string does not contain the substring it returns ‘0’. Because Pos does not have an option to match for whole words, meaning that a search for ‘go’ in the sentence “We’ve got to go” will match the ‘go’ sequence in ‘got’ before it matches the word ‘go’, a ‘while’ loop is used until the keyword is found. Having the ‘while’ loop also allows the function to find collocates if the keyword occurs more than once in the sentence. It does this by looping until all matches for the keyword have been found in the sentence.

When a match is found, the words occurring immediately to the left and right of the keyword are obtained by looping backwards and forwards starting at the keyword. Words are defined as any continuous sequence of letters, including an apostrophe. Once the collocates are found they are stored in vectors called LeftCollocate and RightCollocate. There are also vectors that keep track of how many of each collocate there are, namely LeftCount and RightCount.

After searching through every sentence contained in Concordance, LeftCollocate and RightCollocate will contain all of the collocates of the keyword. The data contained in these vectors is essentially the same as that displayed in Table 4.1 from WordSmith's Collocate program, except that only L1 and R1 are considered.

Next, these collocates are concatenated with the keyword to form phrases. Four pieces of data are needed to do this, corresponding to the vectors created in GetPhrases (Section 4.3.3): the phrase itself, its frequency in the corpus, and whether or not it was formed by adding a stop word on the left and right side of the node.

The elements of both sets of collocates (LeftCollocate and RightCollocate) are iterated in order to form the phrases. I will discuss the loop for LeftCollocate; the loop for RightCollocate is practically the same except that obviously the collocate is concatenated after the keyword rather than before it.

For each element in LeftCollocate (unless its frequency in the corpus is below the cut-off point), exactly one item will be added to the four vectors created in GetPhrases (Section 4.3.3). In GetPhrases the vectors had names like TwoWords and TwoWordsCount, but here they are simply referred to as Phrases, PhrasesCount, and so on. The item added to Phrases is the concatenation of the LeftCollocate and the keyword. The item added to PhrasesCount is the value in LeftCount. The IsStopWord function (discussed in Section 4.3.5) is then called to determine if the collocate is a stop word. IsStopWord returns 'true' if it is a stop word and 'false' if it is not.

If the collocate is a stop word, the value added to PhrasesStopLeft is 'true'; otherwise it is 'false'. The value added to PhrasesStopRight is equivalent to whatever the phrase's previous StopRight value was. This was one of the values passed as a parameter to the function. The entire loop for the left collocates is shown below.

```
for (int i = 0; i < LeftCollocate->size(); i++)
{  if (LeftCount->at(i) >= cut_off)
    {  Phrases->push_back(LeftCollocate->at(i) + " " + Headword);
       PhrasesCount->push_back(LeftCount->at(i));
       //If the newly added left collocate is a stop word,
       //the value for PhrasesStopLeft is true.
       if (IsStopWord(LeftCollocate->at(i).LowerCase(), PHRASE) ||
          (StopWordsForm->To->Checked &&
           LeftCollocate->at(i).LowerCase() == "to"))
       {  PhrasesStopLeft->push_back(true);
       }
       //Otherwise it is false
       else
       {  PhrasesStopLeft->push_back(false);
          left_total = left_total + LeftCount->at(i);
       }
       //The value for PhrasesStopRight is equivalent to the
       //previous phrase's StopRight
       PhrasesStopRight->push_back(is_stop_right);
    }
}
```

To give an example, if the keyword is ‘home’ and the corpus selected is ‘Brown and Frown’, 291 left collocates are found, though only those occurring three or more times are considered. The collocate ‘at’ occurs 139 times to the left of ‘home’, and it is not a stop word. Thus the value ‘at home’ is added to Phrases, ‘139’ is added to PhrasesCount, ‘false’ is added to PhrasesStopLeft, and ‘false’ is added to PhrasesStopRight.

FormPhrases does a similar loop for RightCollocate, then returns.

4.3.5 IsStopWord

The IsStopWord function determines if a word is a stop word, returning ‘true’ if it is and ‘false’ if it is not. There are two different types of stop words used in this program. A stop list is used while creating the index so that high-frequency words are excluded from the index. This greatly reduces both the size of the index and the time it takes to create it as described in Section 4.2.1. A different stop list is also used when searching for phrases. This stop list contains words that are not considered to be important components of expressions, as mentioned in Section 4.3.2. There is some overlap between the two stop lists, but they are by no means identical. The IsStopWord function takes as one of its arguments an integer constant to determine which stop list it should look in. The argument INDEX is used when creating the index while PHRASE is used when building phrases.

The words that are excluded from the index appear in the program listing in Appendix C (Section C.1.9). These are simply some of the most frequently occurring English words. For constancy, they have been categorized, and if the category has a finite number of members, they are all included. For example, although pronouns like ‘I’ occur much more frequently than say ‘we’, all members of the pronoun category have been included.

The stop list used when forming phrases includes many of the same words in the other list, but most notably it does not include prepositions. The use of prepositions is among the most difficult aspects of English for second language learners, so it is important that they be included in the phrases. Otherwise, the exact contents of this stop list can be defined by the user. Selecting Advanced → Stop Words... from the main menu opens the Stop Words form as shown in Figure A.7 in Appendix A. Here, users can select which types of stop words they want to include in the list. By default, the following categories are included: articles (‘the’, ‘a’, ‘an’), conjunctions (‘and’, ‘or’, etc.), subject pronouns (‘I’, ‘you’, etc.), object pronouns (‘me’, ‘them’, etc.), relative pronouns (‘who’, ‘what’, etc.) and possessive adjectives (‘my’, ‘their’, etc.) As mentioned earlier, including articles would result in phrases of the form ‘the ∼’ and ‘a ∼’ to be near the top of the phrase list for virtually any noun the user enters. For this reason they are excluded by default. Article usage is, however, a major concern for many people leaning English, especially those coming from languages that do not use articles (like Chinese and Korean). This program can certainly be used to learn more about article usage, so they can be included if the user wishes. It is a little harder to imagine the usefulness of including conjunctions, pronouns, and possessive adjectives since these seem to be more straightforward. The behavior of pronouns with verbs for example is probably the kind of thing better explained by grammar books than a corpus. Nonetheless the option to remove them from the stop list is always available.

There are also options to include other types of words such as modal verbs and forms of ‘to be’ in the stop list. These do not seem to come up as frequently in the phrases as articles, conjunctions, and pronouns, so they are not included in the stop list by default. All of the categories available to include can be seen on the Stop Words Form in Appendix A (Figure A.7).

The stop list for phrases is stored as a vector in the StopWordsForm class, available publicly. It is created when the Stop Words Form is created, then updated each time the form is closed.

4.3.6 DisplayMessage

As mentioned in Section 4.2, the exclusion of certain high-frequency words from the index means than the user will not be able to search for them. For this reason, if a user enters one of these words, the DisplayMessage function is called in order to tell the user that the word is a high-frequency word and to suggest other types of words that may provide a more useful search. The message display depends on what type of word the user entered. These messages are listed below, with the symbol $\sim$ in place of the word that was entered.

Article “The word $\sim$ is an article and is always used before nouns. For help with using articles, click on Advanced $\rightarrow$ StopWords... from the menu. Uncheck the box next to ‘Articles.’ Then, try searching for various nouns such as ‘people’, ‘time’ or ‘work.’”

Preposition “The word $\sim$ is a preposition and is typically used before nouns and after verbs. For help with using prepositions, try searching for various nouns such as ‘home’, ‘time’, or ‘work’, or for verbs like ‘hold’, ‘put’, or ‘look.’”

Conjunction “The word $\sim$ is a conjunction and is used to join other words or phrases. For help with conjunctions, click on Advanced $\rightarrow$ StopWord... from the menu. Uncheck the box next to ‘Conjunctions.’ Then, try searching for other types of words that are likely to be used with conjunctions.”

Demonstrative “The words ‘this’, ‘that’, ‘these’, and ‘those’ are called demonstratives, and are frequently used with nouns, as in ‘this house’ or ‘those people.’ ‘That’ is also used as a conjunction, as in ‘the house that I live in.’ For help with using demonstratives, click on Advanced → StopWord… from the menu. Make sure the box next to ‘Demonstratives’ is unchecked. Then, try searching for various nouns like ‘house’ and ‘people.’”

Form of ‘be’ “The word ∼ is a form of the word ‘be.’ ‘Be’ is one of the most common words in English. For help with using the various forms of ‘be’, click on Advanced → StopWord… from the menu. Make sure the box next to ‘Forms of “be”’ is unchecked. Then, try searching for other types of words that are likely to be used with ‘be.’”[6]

Pronoun “The word ∼ is a pronoun. Pronouns are generally used with verbs, as in ‘she likes him.’ For help with using pronouns, click on Advanced → StopWord… from the menu. Uncheck the boxes next to ‘Subject pronouns’ and ‘Object pronouns.’ Then try searching for various verbs such as ‘make’, ‘see’, and ‘know.’”

Spoken utterance (like ‘uh’) “Words like ∼ are commonly used in speech to indicate a pause, hesitation, etc. They are not generally part of any expressions.”

‘Not’ “The word ‘not’ is used to make verbs negative, as in “I do not like sushi.” For help with this word, click on Advanced → StopWord… from the menu. Make sure the box next to ‘Negative marker’ is unchecked. Then try searching for various verbs such as ‘make’, ‘see’, and ‘know.’”

‘There’ “‘There’ is one of the most frequently used words in English. It occurs in the expressions ‘there is’ and ‘there are’, as in “there is a small registration fee.” It is also

[6] If the word is ‘be’ itself, the beginning reads “The word ‘be’ is one of the most…”

used as the opposite of 'here.' For help with using 'there', try searching for other words that are commonly used with it."

4.3.7 SortLists

After the OKButtonClick event handler calls GetPhrases, the phrase list is almost ready to be displayed to the user. The only thing left to do is sort it. There are two lists however—one for the phrases themselves, and one for the frequency of the phrases in the corpus. These lists are parallel, meaning that the first value in one corresponds to the first value in the other and so on. Although the C++ Standard Template Library has built-in algorithms for sorting lists, it was necessary to create my own so that two vectors whose contents are parallel can be sorted. This is the purpose of the SortLists function; it sorts the two lists by the values in PhrasesCount, with the highest values first.

The algorithm used is a fairly common one, so I will not go into much detail about it. Essentially, it loops through the vector containing the frequencies (PhrasesCount) comparing neighboring values and swapping them if necessary. It continues this until the lists are sorted.

4.4 TMainForm::ListBoxClick

Another essential part of the program is the ability to click on the ListBox, which is used to display the list of phrases, so that example sentences using the phrase can be displayed. The event handler for ListBoxClick is simple: it just calls the DisplayPhrases function which displays all of the sentences containing the selected phrase. The DisplayPhrases function is also called when the user clicks the OKButton on the Main Form; in this case, the first phrase in the ListBox is automatically highlighted and DisplayPhrases is called with this phrase. A flow chart for this function is available in Appendix B, Figure B.7.

4.4.1 DisplayPhrases

DisplayPhrases is responsible for displaying all the sentences containing the selected phrase in a Rich Edit component called Display. A Rich Edit component is like a mini word processor that can be used to display and edit text in a variety of sizes, colors, and font styles. It uses Rich Text Format (RTF) to code for changes in the appearance of the font, but the component comes with various methods so that it is not necessary for the programmer to deal with RTF encoding. Using these methods, the selected phrase is placed in bold and in blue so that it will stand out in each sentence. DisplayPhrases takes as it argument the phrase that is to be highlighted.

Parts of this function are similar to the FormPhrases function described in 4.3.4. The sentences come from Concordance, a StringList created each time the user clicks the OKButton. It uses the same Pos function to locate the phrase within each sentence, and it also uses a similar type of loop both to ensure that the phrase found is a whole-word match and to find multiple instances of the phrase in a single sentence.

Once a sentence is found that contains the phrase and the position of the phrase in the sentence in known, the sentence is displayed in Display. To change the font property of a portion of text in a Rich Edit component, the portion to be changed must first be selected. The start of the selection is the position of the phrase in the sentence, and the length of the selection is the length of the phrase. (This is done behind the scenes, so to speak; the user does not see any of it.) Then, the font properties of the selected text may be changed to highlight the phrase. A blank line is inserted after each sentence to provide a more appealing layout to the user.

The only tricky part of this function is that the position of the phrase in the sentence is not always equal to the position of the phrase in the Rich Edit component. They are only equal for the first sentence displayed; after that the positions are relative to the previous sentence. For this a variable called where_are_we is used to keep track of the cursor position in Display from sentence to sentence.

4.5 TMainForm::DisplayMouseUp

Technically, the DisplayMouseUp function is called each time the user clicks the Display component, specifically on the release of the button (MouseUp). This event is used so that the user can double click on a particular sentence and view the sentence in its context in the corpus. Another form called Viewer is opened to show the text, with both the sentence and the phrase highlighted (the entire sentence is in blue, with the phrase itself in bold). Unfortunately, Borland does not provide a DoubleClick event for Rich Edit components. Using a single click would be bothersome as it is likely to bring up the viewer at undesired times. For this reason, the single click event (MouseUp) is used to simulate a double click event. It does this by timing the interval between successive clicks, and if the interval is small enough, it calls the GetParagraph function. A flow chart for this function is available in Appendix B, Figures B.8, B.9, and B.10.

4.5.1 GetParagraph

The GetParagraph function locates the sentence that the user clicked on in the corpus, extracts 2400 bytes of surrounding text, and displays it to the user in a form called Viewer. The Viewer form looks a lot like Window's Notepad program, except that it displays text in Rich Text Format (RTF). It is read-only, though it does allow the user to select and copy text as well as save it if desired.

Although GetParagraph seems like a simple function, it is actually a little complicated partly because only the sentences that contain the phrase are saved in memory, meaning that the corpus must be searched to find the full text, and partly because Borland does not provide methods to do certain things that it needs to do.

GetParagraph performs the following tasks:

1. Get the text surrounding the point in Display at which the user clicked.
2. Parse that text into a vector of individual words.
3. Look those words up in the index and make a vector of the index values.
4. Fine the most frequently occurring index values (the modes).
5. Get the byte offsets of the sentence, adding 2400 bytes for surrounding text.
6. Find out which file the text is in and adjust the bytes offsets if necessary.
7. Extract that text from the file.
8. Remove new line and carriage return characters.
9. Verify that it is the correct text, i.e., that it contains the sentence that the user clicked on.
10. Display the text, make the sentence blue, and make the phrase bold.

When the user clicks on Display, the only information available for the program to know where the user clicked is the cursor position. Thus the first step is to use this information to find out which sentence was clicked. To do this, the text of the Display is iterated forwards and backwards starting from the cursor position until it reaches a new line or carriage return character, indicating the beginning or end of the sentence.

Next, the sentence must be located in the corpus. The function takes advantage of the fact that an index for the corpus already exists. Although it requires a little more from the programmer, the result is that the function can locate almost any sentence in the corpus in a fraction of a second. Paradoxically, both extremely short and extremely long sentences take a little longer, for reasons that will be apparent shortly, though it rarely takes over a second even in these cases.

The index contains only words, not phrases, so the sentence must be divided into individual words. Each word is then looked up in the index and its values (which are the numbers of the sentences at which it occurs) are stored in a vector. The values for all the words are stored in a single vector, so that the values in the vector essentially correspond to all of the sentences that might be the one the user clicked on. As most words tend to be reused very frequently in natural language, the vector contains many possible sentences. And since the function did not check to see if the vector already contained a particular value before adding it again, many values will appear more than once. If fact, the more times a value occurs, the more likely it is to be the sentence the user clicked on. For example, if a sentence contains ten words, then the value corresponding to that sentence in the corpus should occur ten times in the vector.[7]

The next step therefore is to find the mode of the vector, that is, the value that occurs the most. To do this, another type of container called a multiset is used, containing the same elements as the vector. A multiset is similar to a vector except that its contents are in no particular order, and it offers different methods for analyzing the data. One of these methods returns the number of times a particular value occurs in the multiset. Thus the value that occurs the most in the multiset is the mode. In some cases, particularly for very short sentences, there may be more than one mode. For example, if the user clicks on a sentence like “I went home”, there is a high probability that another, perhaps longer sentence occurs somewhere in the corpus that also contains those three words. Thus the modes are stored in a vector rather than a simpler variable in case there is more than one mode.

The modes correspond to the sentence numbers in the index, so for each mode the sentence is located in the corpus, in the same way that Concord (Section 4.3.1) locates sentences in the corpus. The only difference is that 1200 extra bytes both before and after the sentence are obtained as well. Next, carriage returns and new-line characters are removed from the text.

[7]Actually, because the index does not include certain high-frequency function words, it would be less than ten times.

Then, using Borland's Pos function, it is tested to see if it contains the sentence that the user clicked on, to make sure that it is the correct text. If it is the correct one, the loop breaks; if not, the loop continues and another block of text is tested. This is why shorter sentences tend to take a little longer to find, since there will be more false matches. Incidentally, longer sentences take longer simply because there are more words to look up in the index.

Once the correct text is found, it is displayed in the Viewer form. The sentence is highlighted using an algorithm similar to the one used in DisplayPhrases (Section 4.4.1).

4.6 TDirectory::OKButtonClick

The OKButtonClick event on the Directory form is the only major event that does not occur on the Main Form of the program. This is no coincidence as I have tried to include as much of the functionality of the program as possible on a single form. This not only helps users learn how to use the program, it also increases the program's ease of use in general, since users do not have to open many other forms to do what they want. In fact, the TDirectory::OKButtonClick is not an event that will occur frequently. The Directory form allows users to create their own corpora by selecting text files from disk. This only needs to be done once; after that, the corpus will automatically be included in the Main Form's SelectText drop-down box.

The Directory form (shown in Appendix A, Figure A.3) uses Windows 3.1 components rather than Windows 95 or later components simply because, in this case, they are actually better suited to the task. The form has three large boxes. The one on the left is the DirectoryListBox; it shows the directory structure of the user's hard drive (or another disk if selected). The box in the middle is called the FileListBox, showing the files in the currently selected directory. And finally, the box on the right is called IncludeFiles; it lists the files that the user wishes to include in the corpus. Two buttons between the FileListBox and

IncludeFiles allow the user to add selected files to or remove selected files from the list. Both boxes support multiple selections.

Clicking the OKButton calls the OKButtonClick event handler. A flow chart for this event handler is shown in Figure B.11, Appendix B. If the user selects only one text, the name of the text, including the full file path, is stored in a class variable called Corpus. This is a public variable available to all other classes in the program. If the user selects more than one text, he or she is prompted to provide a name for the corpus (see Figure A.5). In this case, the value of Corpus is the name the user entered. This name is also used as the name of a file that is created linking all of the texts together.

Before creating this file, any illegal characters not allowed in file names are first removed. In the rare case that nothing is left, the name 'Unnamed' is used. The file is given the extension '.cor.' This file is actually part of the index, as described in Section 4.2.1.

The last thing the event handler does is add the name of the corpus to the SelectText drop-down box on the Main Form. SelectText only contains the name of the corpus (without the complete path of the file). For this reason, in addition to SelectText, the MainForm class also contains a StringList called SelectTextPaths, used in conjunction with SelectText. Thus the full name (including the file path) is added to SelectTextPaths. This way, when the user selects another corpus from SelectText, the corresponding file name in SelectTextPaths can be sent to LoadText, described in Section 4.2.

4.7 Form create and form close events

As mentioned earlier, event handlers are called when the program starts and when the program closes. This allows the programmer to take care of any necessary initialization tasks, and also to save settings. Two initialization files are used with this program: corpora.ini and settings.ini. Corpora.ini lists the corpora that are loaded into SelectText on start-up, as well as the file paths that are loaded into SelectTextPaths. The first time the user loads the program, four corpora will be included in SelectText: the Brown and Frown corpora, the Switchboard corpus, a Wall Street Journal corpus, and an academic corpus (see chapter 3 for more on these corpora). The user may add others to the list by clicking on the FileButton and selecting texts.

Settings.ini includes settings for the stop list and miscellaneous preferences like the status of “Do not show this message again” check boxes. Fortunately, with Borland’s StringList class, very little programming is necessary to make and use initialization files. The StringList class includes functions like LoadFromFile and SaveToFile, and can automatically extract the name and value from a string of the form ‘name=value’. I have thus made use these functions with the initialization files.

Chapter 5

Using and testing the program

Because Phraser finds phrases based on frequency, its results are not limited to any particular type of phrase. This is an advantage over programs like the Cobuild Concordance, which requires users to have an idea of what they are looking for in advance. A few examples of the types of phrases the program is likely to find are: verb-preposition pairs or phrasal verbs ('look at'), adjective-preposition pairs ('similar to'), verb-noun pairs ('take place'), verb-adjective pairs ('make sure'), adverb-adjective pairs ('highly unlikely'), as well as longer phrases like 'get rid of' and 'keep in mind.' Idiomatic expressions like the ones often found in phrase books ('under the weather') do not occur very frequently, at least not in the corpora I have used. While such expressions may add a certain color or charm to one's speech, they are not essential for most forms of interaction. The program thus focuses on what is probably the most important part of the language—the part that is neither too common, like the phrases filtered out by the stop list, nor too odd or obscure, like the phrases that do not show up frequently in a corpus.

Since Phraser was designed to be a flexible tool for all types of phrases, I will show how it can be used by focusing on the results of searches for four different words. An alternative approach would be to focus on certain types of phrases and show how the program can be used to find them. This approach would be well suited for a program like the Cobuild Concordance, but for this program it does not demonstrate its full capacity. The other problem with focusing on types of phrases is that learners do not always categorize phrases like linguists do. Most learners do not stop and wonder "what types of prepositions can I use with the word 'give'?" A more likely question would simply be "how do you use 'give'?", a question that can be more easily answered with a program like this one.

5.1 Example Searches

To show some examples of the types of phrases this program finds, I will discuss the results of four different searches. The words used are 'hold' (Section 5.1.1), 'mind' (Section 5.1.2), 'real' (Section 5.1.3), and 'hand' (Section 5.1.4). These words were selected because they result in a variety of different types of phrases and because they have a number of different uses depending on the context in which they are used.

5.1.1 'Hold'

The resulting phrase list for a search for the word 'hold' in the Brown and Frown corpora is listed in Tables 5.1 and 5.2. Table 5.1 shows the search without any stop words in effect; Table 5.2 shows the same search with all of the possible types of stop words included. These are the same lists that are displayed in the Main Form's List Box. To save space, however, I have listed the phrases in several columns. The user can click on each of these phrases to see a concordance. The user can control the contents of the stop list by selecting Advanced → StopWords... from the main menu as described in Section 4.3.5.

Looking at Table 5.1, the need for a stop list is immediately apparent. Many of the phrases included are due to the fact that extremely high-frequency words like articles and conjunctions tend to collocate with almost every word. Thus since 'hold' is a verb, it is not surprising that the top phrase is the verb with the infinitive marker 'to.' The next two are 'hold' followed by the articles 'the' and 'a.' The same is likely to be true of any transitive verb, since articles frequently occur before the verb's direct object, as do possessive adjectives as in 'hold my.'

Table 5.1: Resulting phrases from a search for 'hold', with no stop words in effect (Brown and Frown Corpora)

to hold (102)	would hold (6)	who hold (3)
hold the (35)	hold him (6)	they hold (3)
hold a (22)	hold of the (6)	his hold (3)
hold on (19)	took hold (5)	on hold (3)
to hold the (17)	not hold (5)	still hold (3)
hold of (14)	a hold (4)	hold its (3)
hold it (13)	can hold (4)	hold still (3)
hold them (11)	hold out (4)	enough to hold (3)
and hold (10)	hold onto (4)	seemed to hold (3)
hold his (10)	hold this (4)	trying to hold (3)
will hold (9)	hold their (4)	to hold down (3)
i hold (9)	hold her (4)	to hold him (3)
to hold a (9)	hold and (4)	to hold back (3)
you hold (8)	to hold up (4)	i hold my (3)
hold up (8)	to hold his (4)	get hold of (3)
hold your (8)	to hold your (4)	don't hold with (3)
we hold (7)	hold on the (4)	got hold of (3)
hold that (7)	to hold on to (4)	take hold of (3)
hold down (7)	taken hold (3)	hold your fire (3)
hold in (7)	which hold (3)	hold in the (3)
to hold them (7)	that hold (3)	View all occurrences of "hold" (303)
could hold (6)		

Table 5.2: Resulting phrases from a search for 'hold', with all stop words in effect (Brown and Frown Corpora)

hold on (19)	hold out (4)	seemed to hold (3)
hold of (14)	hold onto (4)	trying to hold (3)
hold up (8)	taken hold (3)	get hold of (3)
hold down (7)	on hold (3)	got hold of (3)
hold in (7)	still hold (3)	take hold of (3)
took hold (5)	hold still (3)	hold your fire (3)
hold back (5)	hold with (3)	View all occurrences of "hold" (303)
hold on to (5)	enough to hold (3)	

Table 5.2 shows the effectiveness of the stop list in eliminating such phrases. In this case, the list includes phrases that second language learners are likely to find more useful. These include phrasal verbs, which are made up of a verb plus a preposition like ‘on’ or an adverb like ‘down.’ Examples from the list are ‘hold on’, ‘hold up’ and ‘hold down.’ The list also includes frequent verb-noun combinations, with ‘hold’ being used as a noun in this case. Examples include ‘took hold’ and ‘get hold of.’ There is also a verb-adjective combination (‘hold still’) as well as others (‘still hold’, ‘trying to hold’, ‘hold your fire’). Thus the program does not focus on any one type of phrase, but instead emphasizes frequency of occurrence in the corpus. The user does not need to be aware of the fact that ‘hold’ can be used as both a verb (as in ‘hold out’) and a noun (as in ‘get hold of’) since the program will find examples of both if they exist in the corpus. This is an advantage, especially for English, since words in English are seldom confined to a single grammatical category. Even new words have a tendency to expand their grammatical categories from their origin, as in ‘she I.M.’d me.’ Unlike the Cobuild Concordance, this program allows the user to focus on word usage rather than grammatical function.

Aside from the phrase list, the essential part of the program is the concordance. Figure A.2 in Appendix A shows the concordance for the phrase ‘hold on’ from a screen shot of the program. Compare this to the KWIC concordance of the same phrase, as adapted from WordSmith in Figure 5.1. The whole-sentence format used in Phraser is clearly better suited to second language learners than the KWIC format used in WordSmith. The KWIC format focuses on the immediate context, that is on the words surrounding the keyword. The whole-sentence format focuses on a larger context and makes it easier to see how the word is being used in the sentence. The whole-sentence format is more natural, since rarely in readings are people interested in only a particular word or phrase. The KWIC format, although invaluable for linguistic and other types of research, is ultimately not a natural way to read. It was designed to be able to notice linguistic patterns, and it is best suited to what it was designed for.

```
         wages. The Generals Continue To Hold On Thailand's military loses a battle
    by his grandmother. She struggled to hold on to her favorite grandson, even in
       Collor de Mello faces a battle to hold on to his job after a congressional
   to get out a cool, poised, "Won't you hold on a second, please", I covered up
    meted out in one analogous instance. Hold on tight. First of all, the six figures
      Accordingly, aristocrats tended to hold on to their land. It has often been
      "For the murder of Jack Wiggins." "Hold on there, Marshal," Fred said.
 British to so disruptively retain their hold on these vital western posts in
   a passion for the polka and wanted to hold on to that. It could not be done
          idealism, despite its powerful hold on the political traditions of our
   knew, so that he quickly released his hold on the goat and pretended to be
        constantly to give the British a hold on this region, from whence they
        The dancer who never loosens her hold on a parasol, begins to feel that it is
  denominations are rapidly losing their hold on the central city. The key to
the Italian tradition of letting singers hold on to their notes, but to restrain them
    pieces. As soon as the fox has taken hold on most of the populace he imports
        gun on the desk, Marshal". "Now, hold on, damn it; I won't"- Red Hogan's
   out he moves, the thinner will be his hold on conclusive evidence, and the
       At least I had been unable to lay hold on the experience of conversion. Try
  only expressing our present emotion. I hold, on the contrary, that we mean to
```

Figure 5.1: Example of a KWIC display of the phrase ‘hold on’ from the Frown and Brown corpora (adapted from WordSmith).

5.1.2 ‘Mind’

For this example, I have included the following types of words in the stop list: articles, conjunctions, subject pronouns, object pronouns, relative pronouns, and possessive adjectives. This is the default content of the stop list unless the user changes the settings. The stop list essentially works as a filter to remove phrases that may not be of use to the user. With these settings, the filter is neither too strict nor too passive. However, since it is not the job of the programmer to decide what the user wants, the contents of the stop list can easily be set as the user wishes.

The result for a search for ‘mind’ with the default settings is shown in Table 5.3. ‘Mind’ is another word like ‘hold’ that can used in more than one part of speech and has several different meanings depending on its context. One can quickly see the variety of uses for the word in phrases like ‘in mind’, ‘never mind’, ‘don’t mind’, and ‘change her mind.’ Many

Table 5.3: Resulting phrases from a search for ‘mind’ (Brown and Frown Corpora)

in mind (72)	mind in (6)	american mind (3)
of mind (41)	mind for (6)	wouldn’t mind (3)
in his mind (31)	frame of mind (6)	jack’s mind (3)
mind that (26)	would you mind (6)	mind on (3)
of the mind (21)	public mind (5)	mind does (3)
never mind (17)	own mind (5)	mind until (3)
don’t mind (17)	mind from (5)	mind about (3)
to mind (16)	did not mind (5)	mind had (3)
mind of (15)	changed his mind (5)	mind are (3)
mind was (14)	of her mind (5)	goal in mind (3)
mind to (12)	of the human mind (5)	that in mind (3)
mind is (12)	machine in the mind (5)	change his mind (3)
of his mind (12)	made up his mind (5)	before his mind (3)
mind as (11)	scientific mind (4)	came to mind (3)
have in mind (11)	didn’t mind (4)	on my mind (3)
in mind that (11)	won’t mind (4)	up your mind (3)
state of mind (10)	mind were (4)	through her mind (3)
keep in mind (10)	independence of mind (4)	changed her mind (3)
of my mind (10)	of mind that (4)	mind and heart (3)
had in mind (9)	through his mind (4)	to bear in mind (3)
in the mind (9)	change her mind (4)	to keep in mind (3)
in my mind (9)	up her mind (4)	science of the mind (3)
up his mind (8)	bear in mind that (4)	up his mind that (3)
on his mind (8)	keep in mind that (4)	corner of his mind (3)
in her mind (8)	in the public mind (4)	doubt in my mind (3)
bear in mind (7)	out of my mind (4)	in his own mind (3)
human mind (6)	whose mind (3)	if you don’t mind (3)
mind at (6)	york mind (3)	View all occurrences of ”mind” (578)

of the phrases take the form of preposition + 'mind', though even these show a variety of senses, as can be seen from example sentences. 'In mind' tends to be used for what people are thinking, often showing the thoughts that lead up to or could lead up to a particular decision or action. Examples include:

> Rhode Island is going to examine its Sunday sales law with possible revisions **in mind**.
>
> Without any definite plan **in mind**, she went to a judge to see what could be done.
>
> In recognition of the growing trend for second homes, or vacation cottages, we have designed this one specifically with the family handyman **in mind**.

'In the mind', unlike 'in mind' refers more literally to the mind itself. Examples include;

> Scott Fitzgerald said it is a sign of genius to be able to entertain **in the mind** two mutually contradictory ideas without going insane.
>
> The machine **in the mind** offers a more muscular approach.
>
> The town itself, whatever the source of the individual characters, is, as a fictional setting, a metaphor for the female place **in the mind**.

'To mind' may also involve thoughts, but more often it is used with images, remembrances, associations, or sudden ideas. Examples include:

> The words of Cardinal Newman come forcibly **to mind**: "Oh how we hate one another for the love of God"!
>
> The subject of immortality brings **to mind** a vivid incident which took place in 1929 at Montreux in Switzerland.
>
> And when I think about it, the words from a song in a minor Broadway musical, 'Salvation,' come **to mind**.

'To mind' may also occur as verb phrase, as in "Anta, his wife, never seemed **to mind**."

The phrases found with 'on' always occur with a possessive adjective, as in 'on his mind' and 'on my mind.' These are often used for things that are worrying people, for example:

> Winston was relieved; those presents had been **on his mind**.
>
> In spite of the hundred things he had **on his mind**, Winston went and put his arm around her waist.
>
> "I got a lot **on my mind** right now and there are things I can't carry to the field," Davis said.

The example sentences show enough of the context that the non-native speaker of English can use them to help understand the meaning of these phrases. For example 'revisions' and 'definite plan' suggest a certain sense of 'in mind', while 'relieved' and 'in spite of the hundred things' suggest a completely difference sense for 'on... mind'.

5.1.3 'Real'

Part of what makes this program a valuable tool for all types of learners is the ability to use it with specialized corpora. This way, learners can focus on a particular style or register of English in which they would like to become more proficient. Certainly each register of a language has its own vocabulary that a learner must master. But many common words take on different meanings and may be used differently depending on the formality of the situation, the register, and so forth. This is something that shows up very well in this program. The word 'real' is a good example. It is a word that occurs frequently in all four of the corpora used—the Brown and Frown corpora (joined), the Switchboard corpus, the Wall Street Journal corpus, and the academic corpus—but it used differently in each one.

In the Brown and Frown corpora, common phrases with 'real' include 'real estate', '(in) real life', 'very real', 'real world', 'real problem', and '(there was) no real.' In these phrases and in the example sentences, 'real' is typically used to make things concrete, to specify that one is talking about reality and not something fanciful, imaginary, delusional, etc. For example:

> Everyone knows that private detectives **in real life** are not like Sam Spade and Pat Novak, but the real and the imaginary musician are closely linked.
>
> Our problem, therefore, is to devise processes more modest in their aspirations, adjusted to the **real world** of sovereign nation states and diverse and hostile communities.
>
> American Catholic colleges and universities are, in a **very real** sense, the product of “private enterprise”- the “private enterprise” of religious communities.

‘Real estate’ of course has a separate meaning of its own. The second language learner is likely to conclude from these examples that ‘real’ is the opposite of ‘imaginary’ or ‘false’, which is probably the prototypical definition for most people. The Switchboard corpus, however, gives a very different picture of ‘real.’ Here, ‘real’ is more commonly used as an intensifier, as shown in phrases like ‘real good’, ‘real nice’, and ‘real hard.’ Some examples include:

> So that’s still a **real good** show too I that one tends to come on earlier in the day than I want to turn the TV on.
>
> So I think that’s **real nice** too to come up with different options do you like the job sharing.
>
> Just brownies or French doughnuts would have been good but it’s **real hard** to make them they don’t really come out like they do in New Orleans up here I don’t know why.

A learner wishing to become more fluent in informal, conversational English will quickly notice this additional use of ‘real’ in the spoken corpus. The phrases ‘real estate’ and ‘real world’ also occur, showing that the other sense of real has not become obsolete in the spoken data. Sometimes, however, the two senses seems to blur, as in these examples for the phrase 'have a real problem':

> Yeah and and and I **have a real problem** with the government you know giving away things to the the other countries that have as as much ability to to do harm to us as anyone.
>
> And I **have a real problem** with anything pesticide like pesticides or anything like that so.

In these sentences, 'real' is not so much the opposite of 'imaginary' as it is just an intensifier, similar to saying 'big problem.'

From the academic corpus, the top phrase is 'real world', while other phrases show 'real' being used in a way that is more peculiar to certain academic fields: 'real time', 'real numbers', and 'real wages.' From the Wall Street Journal corpus, it is not surprising that 'real estate' dominates the phrase list. Aside from 'real estate' itself there are 'real estate investment' and 'of real estate mortgage.' The phrase 'real time' also occurs here, as in:

> The chief financial officer of the firm was in continuous contact with all of the credit officers providing them with **real time** cash balances information.

These examples show the usefulness of the academic corpus and the Wall Street Journal corpus as sources for information on a specialized form of language. Such corpora are a very valuable resource for virtually all non-native speakers of English and, in some cases, even for native speakers. No matter what one's proficiency in a language, there are always areas for which the usage of certain words can be quite different from what one is used to. Specialized corpora are a valuable tool for people wanting to "learn the lingo" so to speak.

5.1.4 'Hand'

One thing that will become apparent to users of Phraser is that the meanings of words are largely determined by their context. Many times, what a word means in isolation is very different from the way it is normally used in context. This is an important concept because words are seldom used in isolation; they are nearly always used with other words. A good example of this is ‘hand.’ By itself, the word seems very straightforward—it is just the name of a body part. Phraser, however, emphasizes another view of ‘hand’ that is more important for learners of English. The top phrase in all four corpora is ‘on the other hand.’ This phrase almost never refers to the hand itself, but uses ‘hand’ metaphorically. This is also one of the rare examples where a four-word phrase occurs at the top of the list. Phraser is able to do this because the shorter phrase ‘other hand’ (or ‘the other hand’) almost always occurs in the expression ‘on the other hand.’ Therefore, it eliminated the shorter phrase in favor of the complete one.

The phrase’s high occurrence in all four corpora will emphasize for users its importance in a variety of different styles of English, both conversational and formal. Some examples from each corpus are:

> The hall, **on the other hand**, appeared lifeless and deserted on these long waterfront afternoons. (Brown and Frown)
>
> Yeah now my roomie **on the other hand** he is a power user. (Switchboard)
>
> Living organisms, **on the other hand**, cannot stay the same without changing constantly, and they use their environment to their advantage. (Academic)
>
> **On the other hand**, a small decline in gold prices can wipe out profits. (Wall Street Journal)

Other phrases show that the metaphoric use of ‘hand’ is not restricted to this one phrase. Another common phrase that occurs in all four corpora is ‘right hand’ and/or ‘left hand.’ For these it is important to look at the example sentences, since these phrases appear to use hand in its literal body-part sense. Occasionally they are used in this way as in “When her **right hand** was incapacitated by the rheumatism, Sadie learned to write with her **left**

hand." More often, however, the phrases are synonymous with 'right' and 'left' respectively, often used to specify the side of an object, even though the object probably does not have hands. An example is "The **left-hand** numbers in each cell represent utility indicators or net payoff values for A, the **right-hand** numbers those for B." With examples like these, users can note that in the metaphoric use of the phrase, the words are usually hyphenated. But more importantly, these and other phrases show that 'hand' is a very common word in English and occurs in a wide variety of expressions.

There are also some differences in the phrases found in each corpus. The phrases 'hand in' and 'hand over' occur frequently in the spoken corpus and in the Brown and Frown corpora, but do not show up in the Wall Street Journal or academic corpus. 'Out of hand' is one of the top phrases in the spoken corpus, but occurs toward the bottom of the list in the other corpora. Since users can easily switch from one corpus to another using the SelectText box, this program is particularly good at noticing subtle differences in word usage among various corpora. This type of data is merely observational of course; for linguistic research, much more carefully designed and selected corpora would be necessary, and a program like WordSmith would be better for doing statistical analyses on the results. Users of Phraser are more likely to focus on a single corpus that they are interested in. I point out these differences to show how each corpus can emphasize a different style or register of English, and how these differences show up in the search results from this program.

5.2 Testing the program

To test the usefulness of Phraser, I selected two ESL students to answer a short quiz using the program. One is a Brazilian student who has been studying English for less than a year. Thus she does not speak English fluently, even though she is able to read and understand a little. The other is a Korean student who has studied English for many years. She speaks English well, but still makes occasional errors, especially in writing.

The quiz consists of thirteen questions, with the easiest questions toward the beginning. The complete quiz in giving in Appendix D. The questions on the quiz are intended to evaluate three different things: the usefulness of the program's phrase list, the usefulness of the sentences, and the user's understanding of how to use to program.

Questions 1, 2, 3, and 5 are intended to evaluate the usefulness of the phrase list. These questions do not require the user to look at the example sentences. For example, question 1 asks the user to select the appropriate word for the following sentence:

> I was looking for something different in my career, so I decided to ________ advantage of the new opportunities.

The choices given are 'make', 'take', 'get', and 'find.' To find the answer, the user only needs to consult the list of phrases for 'advantage' and notice that 'take advantage' is one of the phrases. Questions like this one test the program's ability to help users select the right word when the surrounding words more or less dictate which word should be used.

Questions 4, 6, 8, 9 11, 12, and 13 require the user to try to infer the meaning of the phrases from the sentences, though 6 can be reasonably guessed by only looking at the phrases. For example, question 8 asks the user to try to determine when to use the expressions 'in mind', 'to mind' and 'on my mind.' These questions test the program's ability to help users learn how to use particular words and assume that they are already able to read and understand the example sentences.

Questions 7 and 10 are related to the actual use of the program. Question 7 asks the user to search for the word 'apart' in the Switchboard corpus, from which the phrases 'falling apart', 'fell apart' and 'is falling apart' are found. The question is how the user can find out if the word 'apart' is used in other expressions. The answer is by either clicking on 'View all occurrences of "apart"' or by searching for 'apart' in another corpus. Question 10 asks the user to search for the word 'wrong' and notice the phrases 'what's wrong' and 'what's wrong with'. The question is how to find out if the phrase 'what's right (with)' is also common. This requires the user to search for the word 'right'.

5.2.1 Results

The beginning ESL student did not understand English well enough to read the quiz, so I had to help her in Portuguese. I did not translate any of the words she looked up, however, nor any word or phrase that she found in the program. If the question was a fill-in-the-blank, I did not help her with any words in the sentence. Despite her limited use of English, she was able to answer questions 1, 2, 3, and 5 successfully. The questions which required her to use the example sentences were more difficult however. The only ones she got right were 9b and 13. Question 9b was to find an expression with the word 'keep' that fits in the sentence:

I had to run in order to ________________ them.

She was able to infer from the example sentences what the expression 'keep up with' meant and saw that it fit in the sentence. Question 13 was to compare the usage of the word 'quarter' in the academic and the Wall Street Journal corpora. She was able to see that 'quarter' in business English typically refers to a fiscal quarter, as in 'fourth quarter earnings.' For the rest of the questions however, she said that there were too many words she did not understand to be able to tell what the sentences meant.

Questions 7 and 10 were somewhat confusing for her. For question 7, she did not realize that 'falling' and 'fell' were related, so her answer was "by clicking on the other expressions." For question 10, to find out if 'what's right (with)' is a common expression, she tried entering the entire expression rather than just the word 'right.'

Phraser turned out to be a much more useful program for the more advanced ESL student. Unlike the beginning student, she knew enough English to be able to read the example sentences from the program and interpret how the phrases were being used. Some of the questions on the quiz were too easy for her, namely 1, 2, 3, and 5. I encouraged her, however, to try to use to program to find the answer, even though for these questions she already knew what the answer was. For the rest of the questions however, the program appeared to be very helpful to her.

She answered all but one the questions correctly. The one she got wrong was number 4, which was a fill-in-the blank question:

> I can't find my camera. Can you help me look _______ it?
>
> A. at B. like C. for D. up

Her initial response was A 'at.' I asked her to look more closely at the example sentences in the program. Her confusion was between the choices 'look at' and 'look for.' Eventually she concluded that 'look at' is more of a perception, and 'look for' is more of an action. She was then able to choose C 'for' as the correct answer.

Questions 8 and 9 were the most difficult for her. Question 8 asks about the difference among the expressions 'in mind', 'to mind', and 'on. . . mind', as described in Section 5.1.2. She did not have much of a problem with 'to mind', since she noticed that it tends to occur with 'come' and 'bring', as in 'come to mind' and 'bring to mind'. The difference between 'in mind' and 'on. . . mind' was more difficult, though she eventually concluded that 'in mind' often occurs with plans and 'on. . . mind' often occurs with burdens. For question 9, she had some doubts about the following fill-in-the-blank question, which asks for an expression using 'keep':

> If you work at home, it's important to _____________________ your business expenses.

The answer I intended was 'keep track of', though she responded with 'keep a lid on.' Either answer seems appropriate however, depending on the meaning of the sentence, which admittedly is not clear as it is written.

For question 11, she had no difficulty realizing that 'went wrong' and 'went right' are not related. 'Went wrong' occurs in sentences such as:

They attributed everything that **went wrong** in Russia to German influence and intrigue.

'Went right' occurs in sentences such as:

"And your golden god", said Samuel Burns, "probably **went right** home and poured himself into a boiling bath."

She concluded that 'went wrong' is used when something wrong happens, while 'went right' is most often used to mean 'move directly to somewhere.' She was also able to notice the difference in usage of the word 'real' in the academic and Switchboard corpora, as described in section 5.1.3, as well as the peculiar usage of 'quarter' in the Wall Street Journal corpus.

5.2.2 Discussion

Although these two tests are not intended as formal research into the usefulness of Phraser, they do show some promising indications that the program is worth-while. Both students admitted that they were not very good with computers, and the second was even a little apprehensive about whether she would be able to learn how to use the program. Once she saw it, however, she caught on very quickly and immediately realized it could be useful for her. She was particularly interested in the academic corpus, being a foreign student in the United States. For most students, it is likely to be in the academic world that their language is most often evaluated, so an academic corpus is clearly the most valuable for them. After the test, she even said that if I sell the program some day, she would definitely buy it. Her only complaint was that it was sometimes difficult to locate particular phrases if there was a long list. This may be due to the nature of the quiz, which requires the user to focus on particular phrases, rather than just the most frequent ones. I asked if it would be useful if there was an option to put the phrases in alphabetical order, and she said that would help a lot. That way, users can choose which order is most convenient for them.

The beginning student's reaction to the program was also positive, despite not being able to answer the more difficult questions. She said that she was surprised that she was able to answer as many questions as she did, having only been through beginning English classes for speakers of Portuguese. Rather than being frustrating or intimidating to her, the program actually appeared to make her more confident in her English, since she saw that some questions that appeared difficult were actually very easy using the program (namely the multiple choice fill-in-the-blank questions).

5.3 Further developments

There are many additional features that could be added to this program to make it more useful. The test with the beginning ESL student suggests that the program is of some use for beginning students, but they are unlikely to learn as much from it as a more advanced student. One thing that could make this program more useful for beginning students is the ability to use it with a parallel bilingual corpus. This way, students could compare sentences in English with translations in their own language. Such a feature would not translate words or phrases for the students, since the student could consult a foreign-language dictionary for this. Instead, it would allow students to see that a word in English, like 'mind' or 'hand' in the examples given earlier (Sections 5.1.2 and 5.1.4), is not equivalent to the dictionary definition in their own language, but rather takes on a life of its own depending on how it is used. The translations would merely serve to help the user understand the sentence, which was a problem for the beginning ESL student who tested this program. I suggested the idea of using a bilingual corpus to her, and she said it would help a lot, adding that she would easily be able to answer all of the questions if she knew what the sentences meant.

Many other useful features could be added if the corpora were tagged for parts of search. Only one of the corpora used in this thesis (the Switchboard corpus) has part-of-speech tags. To make full use of them, however, would require a more sophisticated indexing system, or else a completely different method for locating words in the corpus. With the ability to know what part of speech a word belongs to however, the program could do more than just look for collocations; it could also look for colligations, words that are related by a particular grammatical feature. For example, one feature of English that is often difficult for speakers of other languages is knowing when to use the infinitive or the '-ing' form of the verb, as in "She likes to shop" vs. "She likes shopping." The current program would not be of much use for this, however with a tagged corpus it could group all infinitives together as well as all '-ing' forms and show which one is more frequent. For example, a search for 'avoid' would find many examples of 'avoid + -ING', but would probably not find 'avoid + INFINITIVE' (cf. 'avoid going to school' vs. *'avoid to go to school').

Along these same lines, other features could be added if the program grouped words together based on certain grammatical categories, like possessive adjectives, or on lemmas. With the current program for example, a search for the word 'shook' returns phrases like 'shook his head' and 'shook her head.' The program could recognize 'his' and 'her' as possessive adjectives and replace them with a general word like 'one's.' A similar type of thing could be done with lemmas, for example, a search for 'shook' could also search for 'shake', 'shakes', 'shaken', and 'shaking' and group the results accordingly, arriving at generalized expressions like 'shake one's head.' There is some value in leaving the program like it is, however. Phrases like 'shake one's head' are typical of dictionary and phrase book entries, but they rarely occur as such in real language. 'Shook his head' and 'shook her head' are far more typical, and arguably, some learners may benefit more from seeing the phrases as they actually occur. Secondly, many verb phrases are used more in one tense than another, while noun phrases may exist in the singular but not in the plural, or vice versa. For example, a search for 'time' finds phrases such as 'at the same time', 'for the first time', and 'long time', while a search for 'times' results in phrases like 'at times', 'three times' and 'at all times.' At least in this example, it does not appear that the behavior of 'time' is much like that of ‘times’. Examples like these, though, may be more of an exception to the general tendency, and there may be some way for the program to deal with them.

As for the internal algorithms of the program, the multiple search algorithm described in Section 4.3.2 seems to work quite well for finding most types of expressions. Its only major disadvantage is that it does not work well for phrasal verbs in which the phrasal part has been separated from the verb, as in ‘**take** the trash **out**’. A possible solution to this would be to make a full table of collocates, like the one shown in table 4.1 for WordSmith, rather that restricting the view to the collocates immediately to the left and right. This would not be necessary in all of the searches, only the first one for the keyword itself. This way, the program could include sentences with phrases like ‘take the trash out’ in the examples for ‘take out’.

The indexing system for this program also has room for improvement. Ideally, the best method for text retrieval would be one that can search the text reasonably quickly without using an index. Having to create an index is problematic because it may take ten to twenty minutes for even a small corpus, and several hours or more for a large corpus. However, file indexing is a complex and developing field of study in itself, so the best solution to this problem is to find a ready-made text retrieval system to use in this program. These are rarely if ever free for the Windows operating system, however, and I did not have the resources to get one for this project.

Chapter 6

Conclusions

There are no doubt many other features that could be added to Phraser, but the basic framework of the program that has been created already appears to be useful for students learning English. The program is flexible enough that it can be used in many different ways. With a sufficiently large corpus, it could be used as a reference guide to help students who are not sure of how to use a particular word. In this sense, it could be used as another type of dictionary, one that gives examples instead of definitions. It is important to remember, however, that a small or moderately sized corpus like that ones used in this thesis do not contain examples of all possible combinations of words, only the most probable ones. For this reason, I envision this program as being used primarily as a tool for learning. In a classroom, the program could be used in conjunction with other assignments, such as exercises, readings, and compositions. For example, rather than explaining the difference between 'much', 'many', and 'a lot', an instructor could let students use this program and have them "discover" the difference themselves. Since the program automates the task of finding phrases, this would not require a large amount of extra work for the student, nor does it require them to have any training in corpus research. Although the program does not provide the answers to students' questions, it does present the data in a way that makes the answers easy to find.

The tests with both a beginning and an advanced ESL student suggest that the program can be useful to both in some ways, but it is more likely to be useful for students who know enough English to be able to understand words and phrases from their context. Thus the program is not a substitute for a dictionary or phrase book; it is a tool that gives students experience with real language, but in an organized way that allows them to focus on words with which they are having difficulty.

The strength of the program lies in its ability to look at words in the context of other words, not just as individual units with discrete and independent meanings. In this way, it forces the learner to think more along the lines of “how do I use the word X?” rather than “what does X mean?”

Bibliography

Benson, Morton, Evelyn Benson, and Robert Ilson 1997. *The BBI Combinatory Dictionary of English: A Guide to Word Combinations.* Revised Edition. Amsterdam: John Benjamins Publishing Company.

Bernadini, Silvia 2000. "Systematising serendipity: Proposals for concordancing large corpora with language learners." *Rethinking Language Pedagogy from a Corpus Perspective: Papers from the third international conference on Teaching and Language Corpora.* Lou Burnard and Tony McEnery (eds.) Frankfurt am Main: Peter Lang.

— 2002. "Exploring new directions for discovery learning." *Teaching and Learning by Doing Corpus Analysis: Proceedings of the fourth international conference on Teaching and Language Corpora.* Bernhard Kettemann and Georg Marko (eds.) Amsterdam: Rodopi.

Granger, Sylviane 1998. "The computer learner corpus: a versatile new source of data for SLA research." *Learner English on Computer.* Sylviane Granger, ed. London: Longman.

Hunston, Susan 2002. *Corpora in Applied Linguistics.* Cambridge: Cambridge University Press.

Johns, Tim 1991. "Should you be persuaded—Two samples of data-driven learning materials." *Classroom Concordancing.* Tim Johns and Phillip King, eds. ELR Journal, Vol. 4.

Leech, Geoffrey 1998. "Preface." *Learner English on Computer.* Sylviane Granger, ed. London: Longman.

Mair, Christian 2002. "Empowering non-native speakers: The hidden surplus value of corpora in continental English departments." *Teaching and Learning by Doing Corpus Analysis: Proceedings of the fourth international conference on Teaching and Language Corpora.* Bernhard Kettemann and Georg Marko (eds.) Amsterdam: Rodopi.

Mparutsa, Cynthia, Alison Love, and Andrew Morrison 1991. "Bringing concord to the ESP classroom." *Classroom Concordancing.* Tim Johns and Phillip King, eds. ELR Journal, Vol. 4.

Scott, Mike 2001. "Comparing corpora and identifying key words, collocations, frequency distributions through the WordSmith Tools suite of computer programs." *Small Corpus Studies and ELT: Theory and Practice* Mohsen Ghadessy, Alex Henry, and Robert L. Roseberry (eds.) Amsterdam: John Benjamins.

Stubbs, Michael 2001. *Words and Phrases: Corpus Studies of Lexical Semantics.* Malden: Blackwell Publishers.

Wible, David, et al. 2002. "Toward automating a personalized concordancer for data-driven learning: a lexical difficulty filter for language learners." *Teaching and Learning by Doing Corpus Analysis: Proceedings of the fourth international conference on Teaching and Language Corpora.* Bernhard Kettemann and Georg Marko (eds.) Amsterdam: Rodopi.

Witten, Ian H., Alistair Moffat and Timothy C. Bell 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Second Edition. San Francisco: Morgan Kaufmann Publishers, Inc.

Wooldridge, T. Russon 1991. "A CALL application in vocabulary and grammar". *CCH Working Papers, vol. 1.* http://www.chass.utoronto.ca/epc/chwp/wulfric1/index.html.

Corpora and Other Texts Used

Bradon, Edwin Philip 1987. *Do Teachers Care About Truth? Epistemological Issues for Education.* London: Allen & Unwin. http://www.uwichill.edu.bb/bnccde/epb/PREF.html

Buchanan, James M 1975. *The Limits of Liberty: Between Anarchy and Leviathan.* Chicago: University of Chicago Press. http://www.econlib.org/library/Buchanan/buchCv7Contents.html

Cohen, Bernard Leonard 1990. *The Nuclear Energy Option: An Alternative for the 90s.* New York : Plenum Press. http://www.phyast.pitt.edu/∼blc/BOOK.htm

Committee on Issues in the Transborder Flow of Scientific Data et al. 1997. *Bits of Power: Issues in Global Access to Scientific Data.* Washington, D.C.: National Academy Press. http://www.nap.edu/readingroom/books/BitsOfPower/

Darrigol, Olivier. 1992. *From c-Numbers to q-Numbers: The Classical Analogy in the History of Quantum Theory.* Berkeley: University of California Press. http://ark.cdlib.org/ark:/13030/ft4t1nb2gv/

Francis, W.N., and H. Kucera. 1979. *Brown Corpus.* Providence, RI: Brown University. ICAME CD-Rom.

Goertzel, Ben 1993. *The Evolving Mind.* Langhorne, PA: Gordon and Breach. http://www.goertzel.org/books/mind/contents.html

Greenwald, Lloyd N/A. Wall Street Journal Corpus. http://plan.mcs.drexel.edu/courses/ml/software/

Herman, Ellen 1995. *The Romance of American Psychology: Political Culture in the Age of Experts.* Berkeley: University of California Press. http://ark.cdlib.org/ark:/13030/ft696nb3n8/

Hundt, Marianne, Andrea Sand, and Paul Skandera 1999. *The Freiburg-Brown Corpus of American English* Freiburg: Albert-Ludwigs-Universität. ICAME CD-Rom.

Ide, Nancy, and Randi Reppen, Keith Suderman 2003. *Switchboard Corpus*, from *The American National Corpus.* American National Corpus Project, CD-Rom.

Krupat, Arnold 1992. *Ethnocriticism: Ethnography, History, Literature.* Berkeley: University of California Press. http://ark.cdlib.org/ark:/13030/ft9m3nb6fh/

Lodish, Harvey 2003. *Molecular Cell Biology.* New York: W.H. Freeman and Company. http://www.ncbi.nlm.nih.gov/books/

Sahtouris, Elisabet 1996. *Earthdance: Living Systems in Evolution.* Santa Barbara, CA: Metalog Books. http://www.ratical.com/LifeWeb/Erthdnce/erthdnce.html

Sinclair, John, ed. 1995. *Collins Cobuild English Dictionary.* London: Harper Collins.

Stephenson, Neal 1999. *In the beginning...was the command line* New York: Avon Books. http://www.cryptonomicon.com/beginning.html

Von Mises, Ludwig 1979. *Economic Policy: Thoughts for Today and Tomorrow.* South Bend, Ind.: Regnery/Gateway. http://www.mises.org/etexts/ecopol.asp

Appendix A

Screen Shots

Note: Screen shots of all the forms in Phraser are shown in this appendix. Figure A.1 shows the Main Form with the names of the visual components labeled. The other figures show the program exactly as it appears to the user. The Main Form is resizable, and users can maximize it if desired.

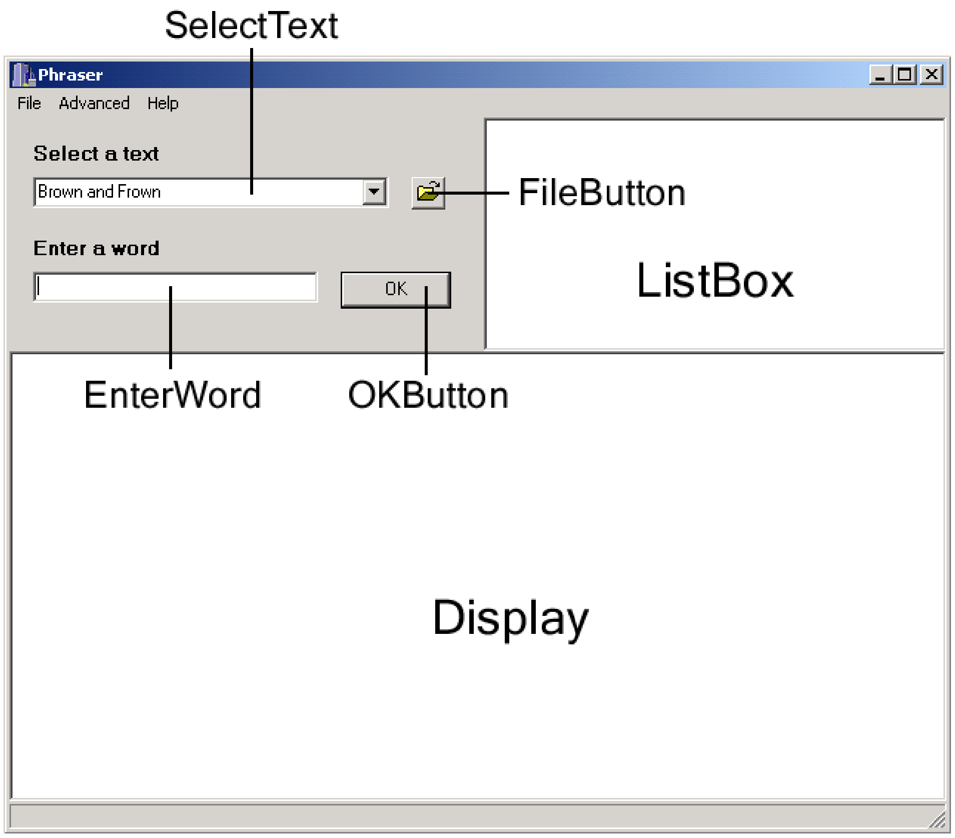
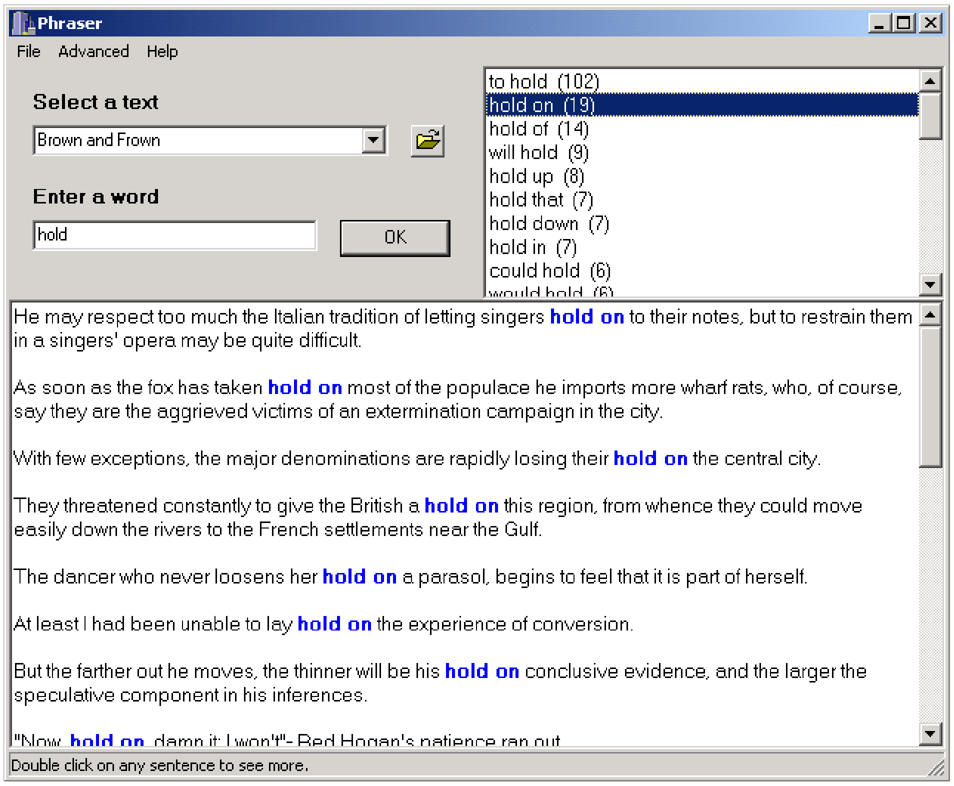


Figure A.1: Screen shot of the Main Form, with names of components labeled.



to hold (102)
hold on (19)
hold of (14)
will hold (9)
hold up (8)
hold that (7)
hold down (7)
hold in (7)
could hold (6)
would hold (6)

He may respect too much the Italian tradition of letting singers **hold on** to their notes, but to restrain them in a singers' opera may be quite difficult.

As soon as the fox has taken **hold on** most of the populace he imports more wharf rats, who, of course, say they are the aggrieved victims of an extermination campaign in the city.

With few exceptions, the major denominations are rapidly losing their **hold on** the central city.

They threatened constantly to give the British a **hold on** this region, from whence they could move easily down the rivers to the French settlements near the Gulf.

The dancer who never loosens her **hold on** a parasol, begins to feel that it is part of herself.

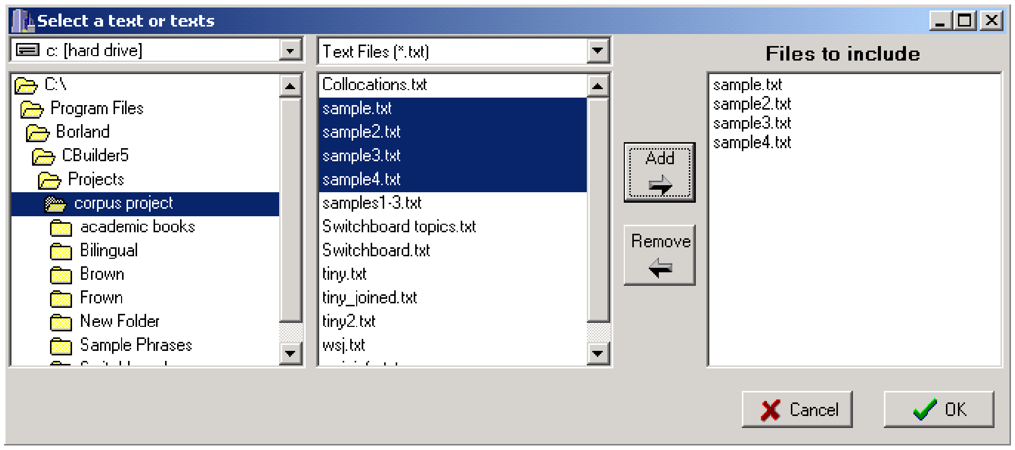
At least I had been unable to lay **hold on** the experience of conversion.

But the farther out he moves, the thinner will be his **hold on** conclusive evidence, and the larger the speculative component in his inferences.

"Now, **hold on**, damn it, I won't"- Red Hogan's patience ran out.

Double click on any sentence to see more.

Figure A.2: Screen shot of the Main Form, with an example search for 'hold.'

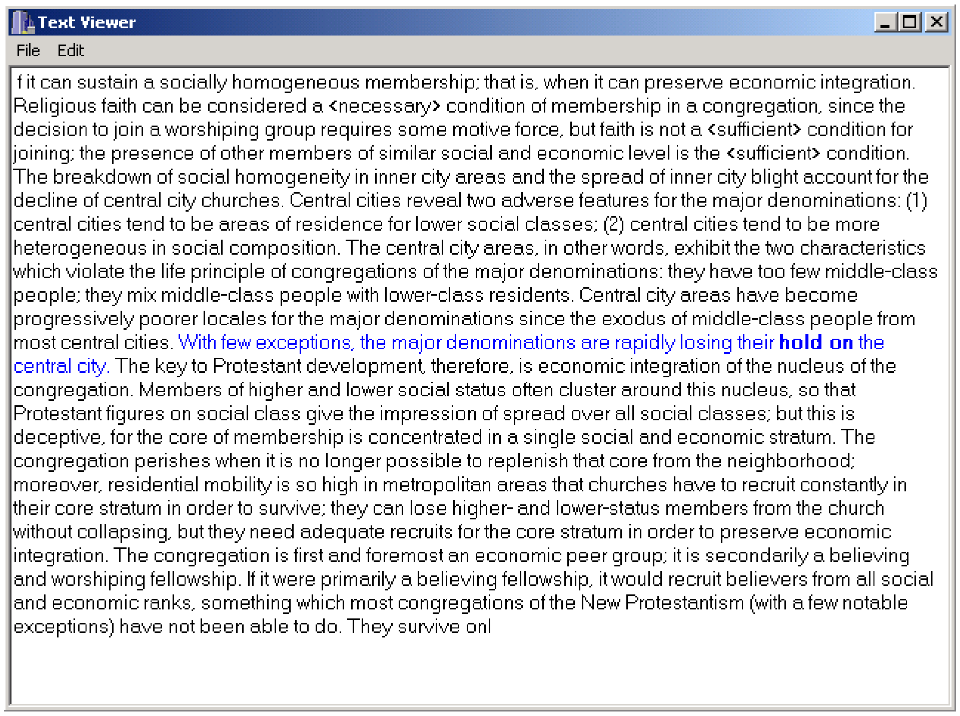


C:\
Program Files
Borland
CBuilder5
Projects
corpus project
academic books
Bilinguial
Brown
Frown
New Folder
Sample Phrases

Collocations.txt
sample.txt
sample2.txt
sample3.txt
sample4.txt
samples1-3.txt
Switchboard topics.txt
Switchboard.txt
tiny.txt
tiny_joined.txt
tiny2.txt
wsj.txt

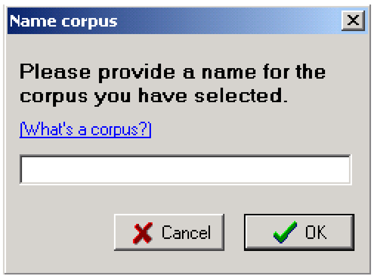
sample.txt
sample2.txt
sample3.txt
sample4.txt

Figure A.3: Screen shot of the Directory form.



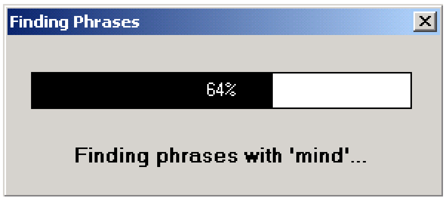
f it can sustain a socially homogeneous membership; that is, when it can preserve economic integration. Religious faith can be considered a <necessary> condition of membership in a congregation, since the decision to join a worshiping group requires some motive force, but faith is not a <sufficient> condition for joining; the presence of other members of similar social and economic level is the <sufficient> condition. The breakdown of social homogeneity in inner city areas and the spread of inner city blight account for the decline of central city churches. Central cities reveal two adverse features for the major denominations: (1) central cities tend to be areas of residence for lower social classes; (2) central cities tend to be more heterogeneous in social composition. The central city areas, in other words, exhibit the two characteristics which violate the life principle of congregations of the major denominations: they have too few middle-class people; they mix middle-class people with lower-class residents. Central city areas have become progressively poorer locales for the major denominations since the exodus of middle-class people from most central cities. With few exceptions, the major denominations are rapidly losing their **hold on** the central city. The key to Protestant development, therefore, is economic integration of the nucleus of the congregation. Members of higher and lower social status often cluster around this nucleus, so that Protestant figures on social class give the impression of spread over all social classes; but this is deceptive, for the core of membership is concentrated in a single social and economic stratum. The congregation perishes when it is no longer possible to replenish that core from the neighborhood; moreover, residential mobility is so high in metropolitan areas that churches have to recruit constantly in their core stratum in order to survive; they can lose higher- and lower-status members from the church without collapsing, but they need adequate recruits for the core stratum in order to preserve economic integration. The congregation is first and foremost an economic peer group; it is secondarily a believing and worshiping fellowship. If it were primarily a believing fellowship, it would recruit believers from all social and economic ranks, something which most congregations of the New Protestantism (with a few notable exceptions) have not been able to do. They survive onl

Figure A.4: Screen shot of the Viewer form, with an example from ‘hold on.’



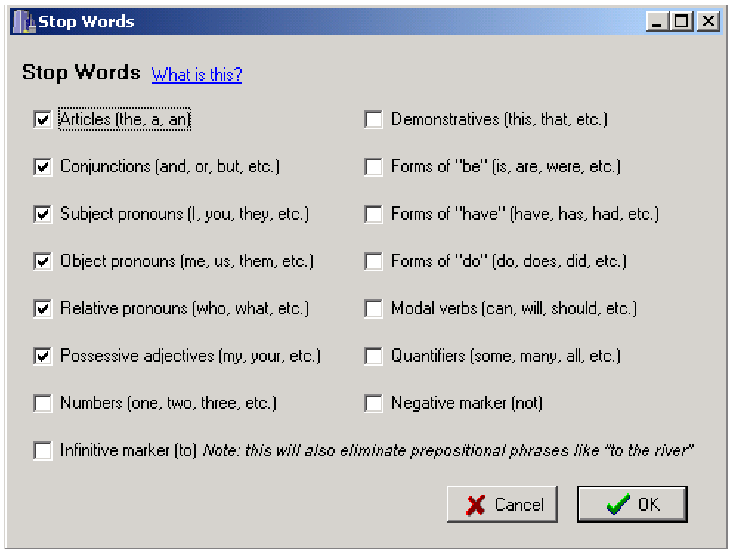
Please provide a name for the corpus you have selected.

Figure A.5: Screen shot of the Name Corpus Form.



Finding phrases with 'mind'...

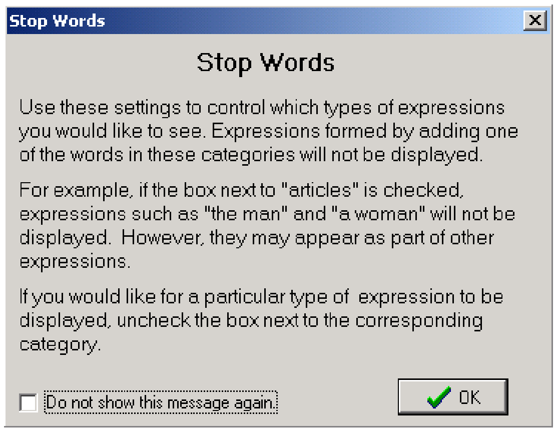
Figure A.6: Screen shot of the Progress Form, searching for ‘mind’.



Stop Words

- [✓] Articles (the, a, an)
- [✓] Conjunctions (and, or, but, etc.)
- [✓] Subject pronouns (I, you, they, etc.)
- [✓] Object pronouns (me, us, them, etc.)
- [✓] Relative pronouns (who, what, etc.)
- [✓] Possessive adjectives (my, your, etc.)
- [ ] Numbers (one, two, three, etc.)
- [ ] Demonstratives (this, that, etc.)
- [ ] Forms of "be" (is, are, were, etc.)
- [ ] Forms of "have" (have, has, had, etc.)
- [ ] Forms of "do" (do, does, did, etc.)
- [ ] Modal verbs (can, will, should, etc.)
- [ ] Quantifiers (some, many, all, etc.)
- [ ] Negative marker (not)
- [ ] Infinitive marker (to) *Note: this will also eliminate prepositional phrases like "to the river"*

Figure A.7: Screen shot of the Stop Words Form.



Stop Words

Use these settings to control which types of expressions you would like to see. Expressions formed by adding one of the words in these categories will not be displayed.

For example, if the box next to "articles" is checked, expressions such as "the man" and "a woman" will not be displayed. However, they may appear as part of other expressions.

If you would like for a particular type of expression to be displayed, uncheck the box next to the corresponding category.

Figure A.8: Screen shot of the Stop Words Message.

Appendix B

Flow Charts

Flow Charts are shown for all of Phraser's major functions and event handlers. They are written using English sentences and pseudo-code rather than C++. For the actual programming code, consult Appendix C.

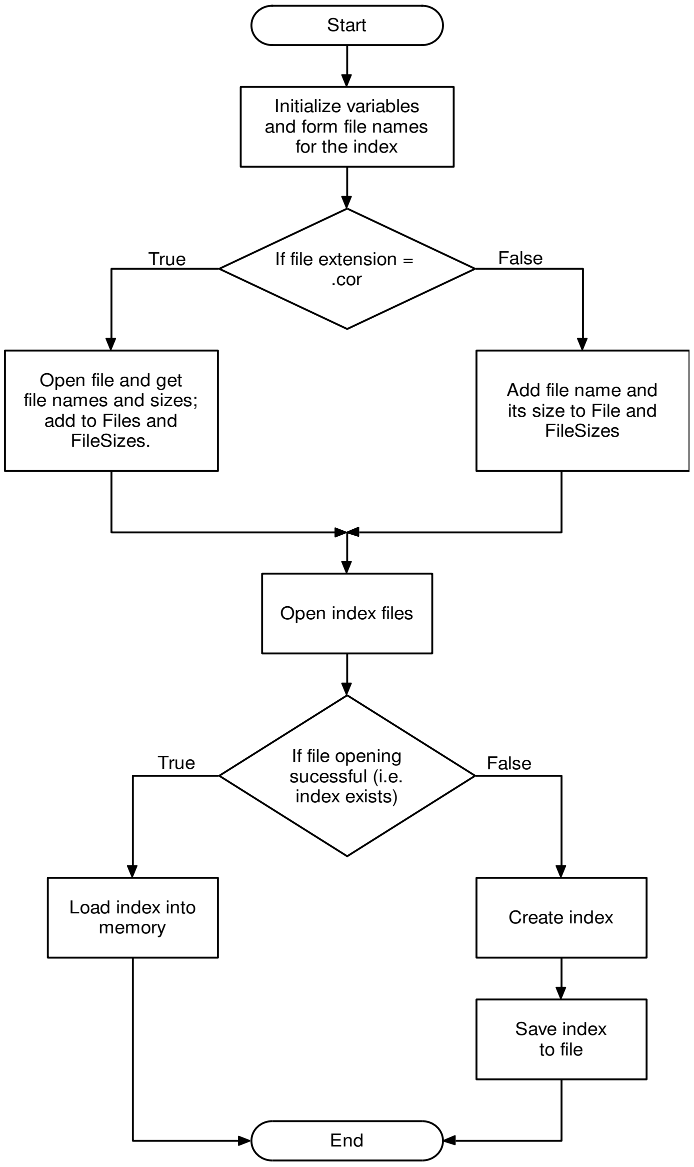

Start
Initialize variables and form file names for the index
If file extension = .cor
True
False
Open file and get file names and sizes; add to Files and FileSizes.
Add file name and its size to File and FileSizes
Open index files
If file opening sucessful (i.e. index exists)
True
False
Load index into memory
Create index
Save index to file
End

Figure B.1: Flow chart for LoadText function.

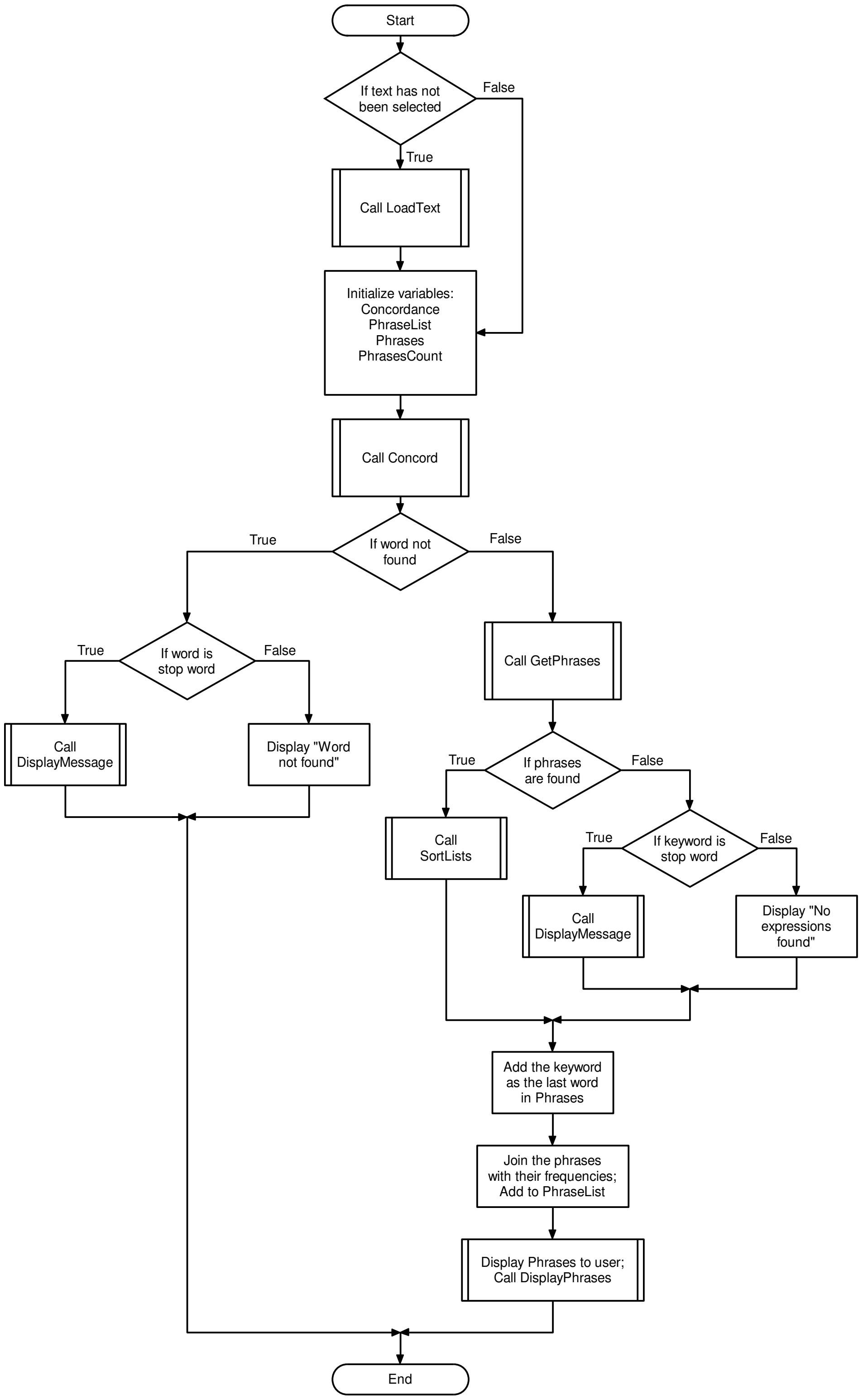

Start
If text has not been selected
False
True
Call LoadText
Initialize variables: Concordance PhraseList Phrases PhrasesCount
Call Concord
If word not found
True
False
If word is stop word
True
False
Call DisplayMessage
Display "Word not found"
Call GetPhrases
If phrases are found
True
False
Call SortLists
If keyword is stop word
True
False
Call DisplayMessage
Display "No expressions found"
Add the keyword as the last word in Phrases
Join the phrases with their frequencies; Add to PhraseList
Display Phrases to user; Call DisplayPhrases
End

Figure B.2: Flow chart for TMainForm::OKButtonClick event handler.

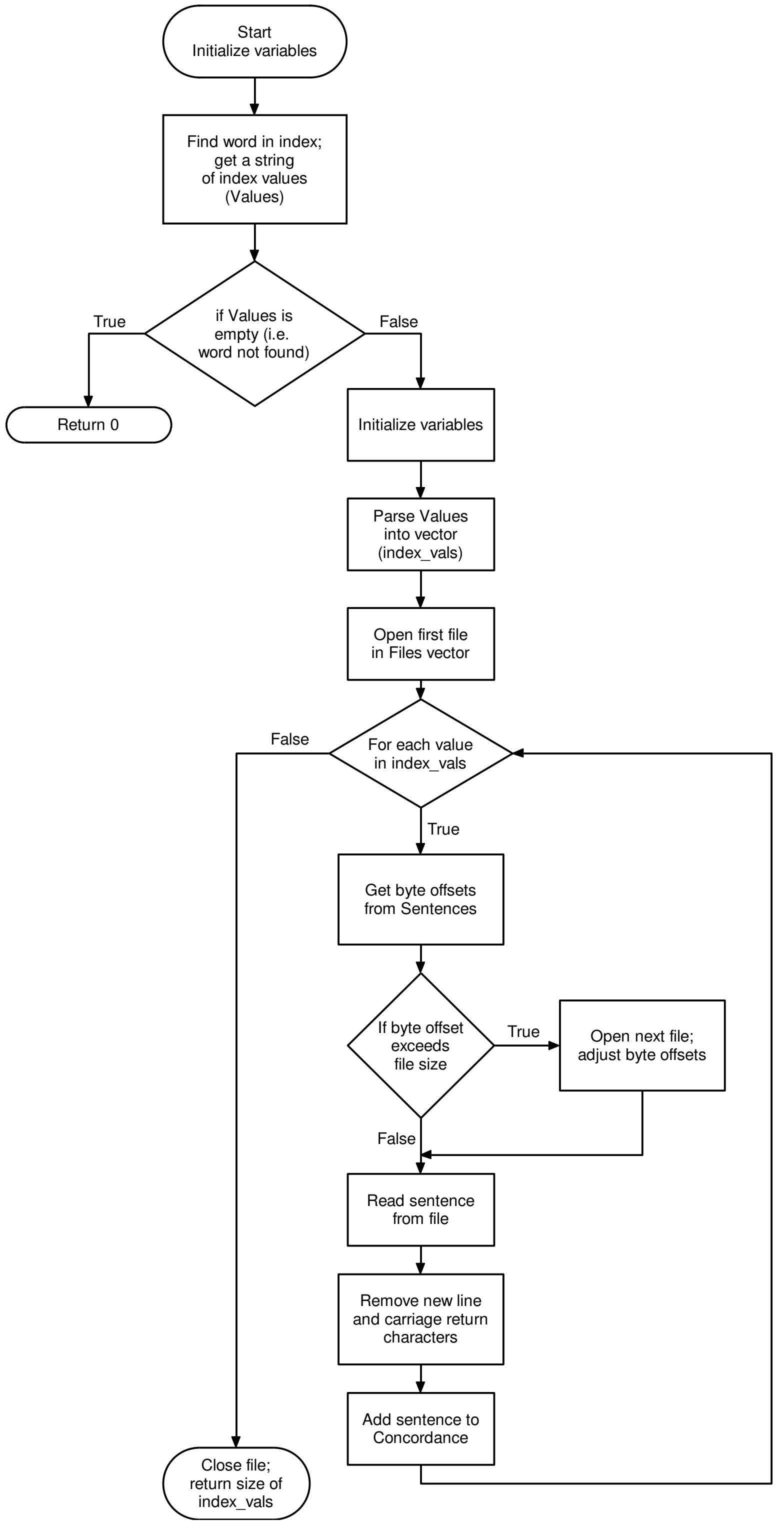

Start
Initialize variables
Find word in index; get a string of index values (Values)
if Values is empty (i.e. word not found)
True
False
Return 0
Initialize variables
Parse Values into vector (index_vals)
Open first file in Files vector
For each value in index_vals
False
True
Get byte offsets from Sentences
If byte offset exceeds file size
True
Open next file; adjust byte offsets
False
Read sentence from file
Remove new line and carriage return characters
Add sentence to Concordance
Close file; return size of index_vals

Figure B.3: Flow chart for Concord function.

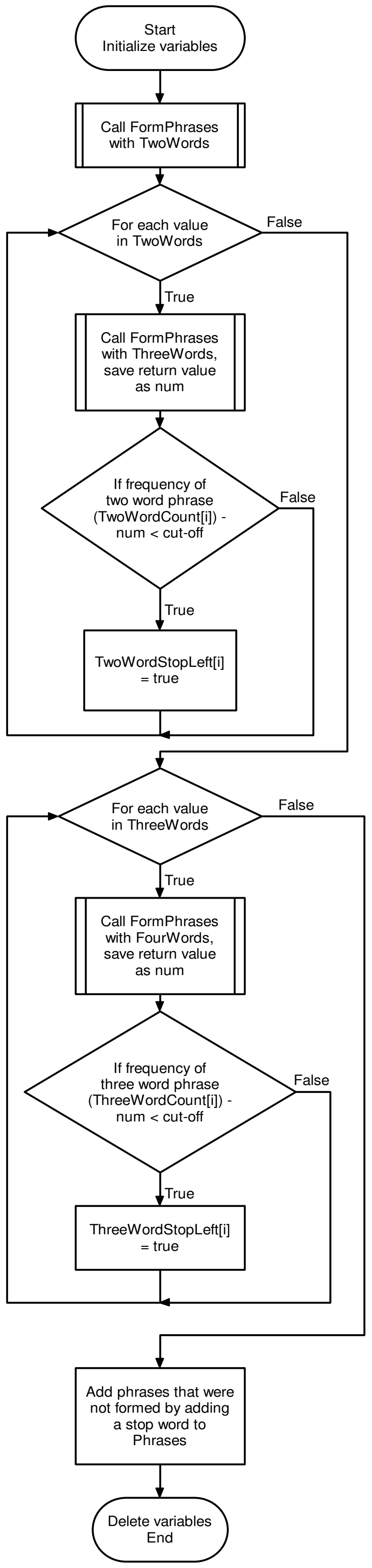
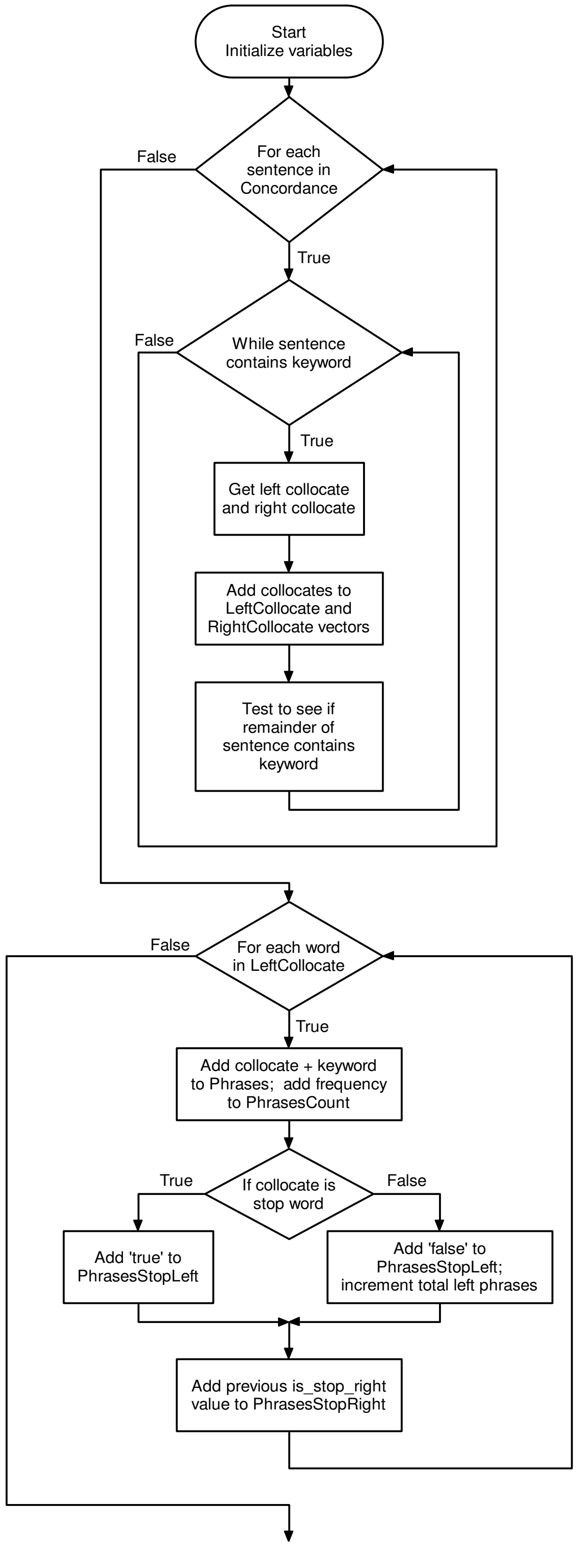

Start
Initialize variables
Call FormPhrases with TwoWords
For each value in TwoWords
False
True
Call FormPhrases with ThreeWords, save return value as num
If frequency of two word phrase (TwoWordCount[i]) - num < cut-off
False
True
TwoWordStopLeft[i] = true
For each value in ThreeWords
False
True
Call FormPhrases with FourWords, save return value as num
If frequency of three word phrase (ThreeWordCount[i]) - num < cut-off
False
True
ThreeWordStopLeft[i] = true
Add phrases that were not formed by adding a stop word to Phrases
Delete variables
End

Figure B.4: Flow chart for GetPhrases function


Start
Initialize variables
For each sentence in Concordance
False
True
While sentence contains keyword
False
True
Get left collocate and right collocate
Add collocates to LeftCollocate and RightCollocate vectors
Test to see if remainder of sentence contains keyword
For each word in LeftCollocate
False
True
Add collocate + keyword to Phrases; add frequency to PhrasesCount
If collocate is stop word
True
False
Add 'true' to PhrasesStopLeft
Add 'false' to PhrasesStopLeft; increment total left phrases
Add previous is_stop_right value to PhrasesStopRight

Continued on next page...

Figure B.5: Flow chart for FormPhrases function (1 of 2)

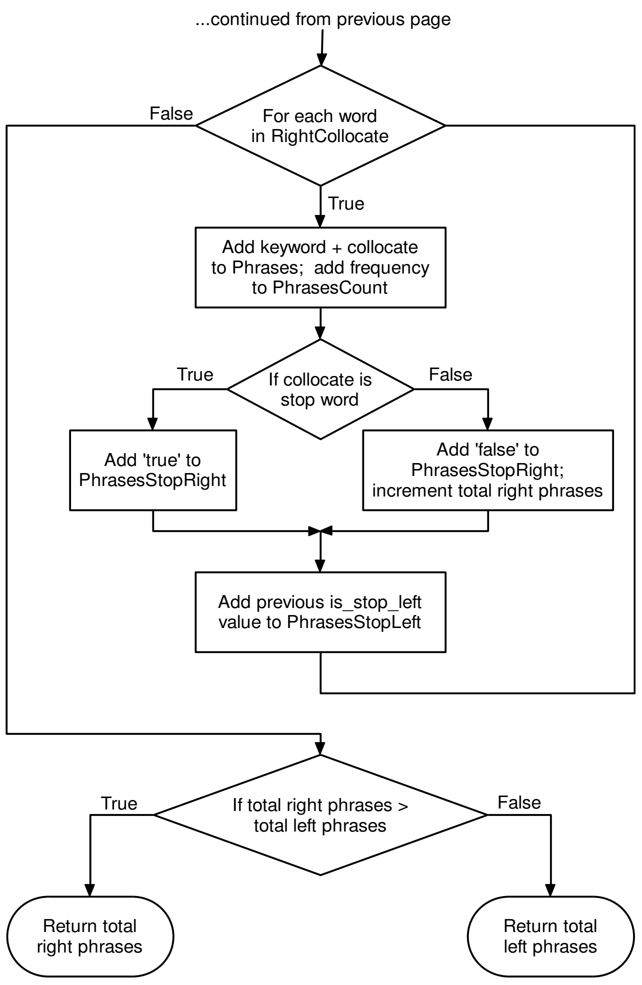

...continued from previous page
For each word in RightCollocate
False
True
Add keyword + collocate to Phrases; add frequency to PhrasesCount
If collocate is stop word
True
False
Add 'true' to PhrasesStopRight
Add 'false' to PhrasesStopRight; increment total right phrases
Add previous is_stop_left value to PhrasesStopLeft
If total right phrases > total left phrases
True
False
Return total right phrases
Return total left phrases

Figure B.6: Flow chart for FormPhrases function (2 of 2)

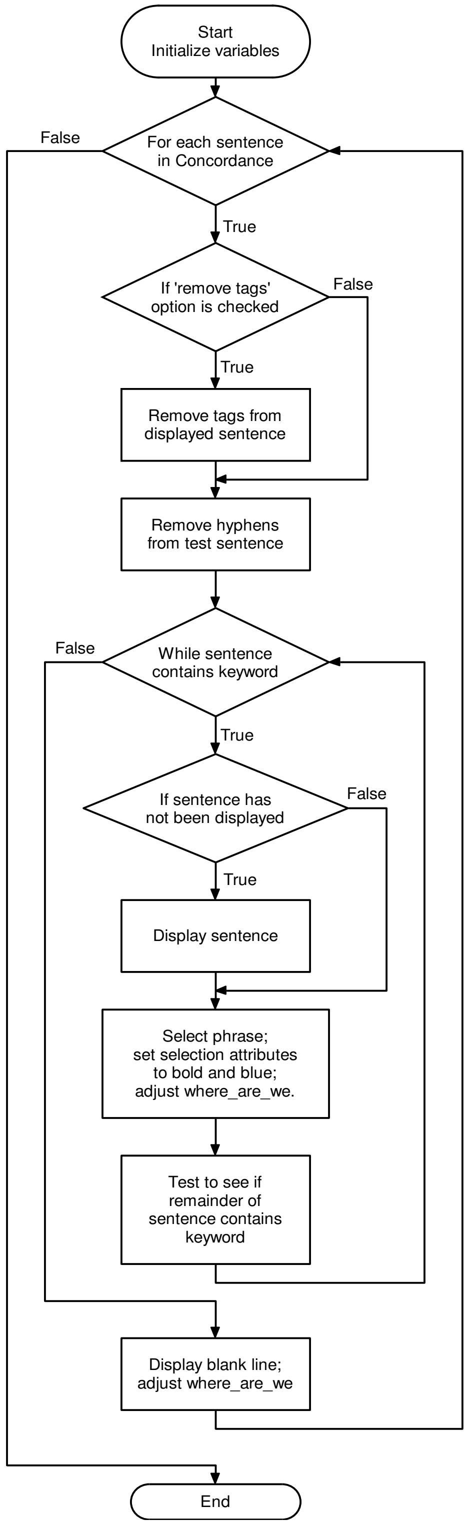

Start
Initialize variables
For each sentence in Concordance
False
True
If 'remove tags' option is checked
False
True
Remove tags from displayed sentence
Remove hyphens from test sentence
While sentence contains keyword
False
True
If sentence has not been displayed
False
True
Display sentence
Select phrase; set selection attributes to bold and blue; adjust where_are_we.
Test to see if remainder of sentence contains keyword
Display blank line; adjust where_are_we
End

Figure B.7: Flow chart for DisplayPhrases function

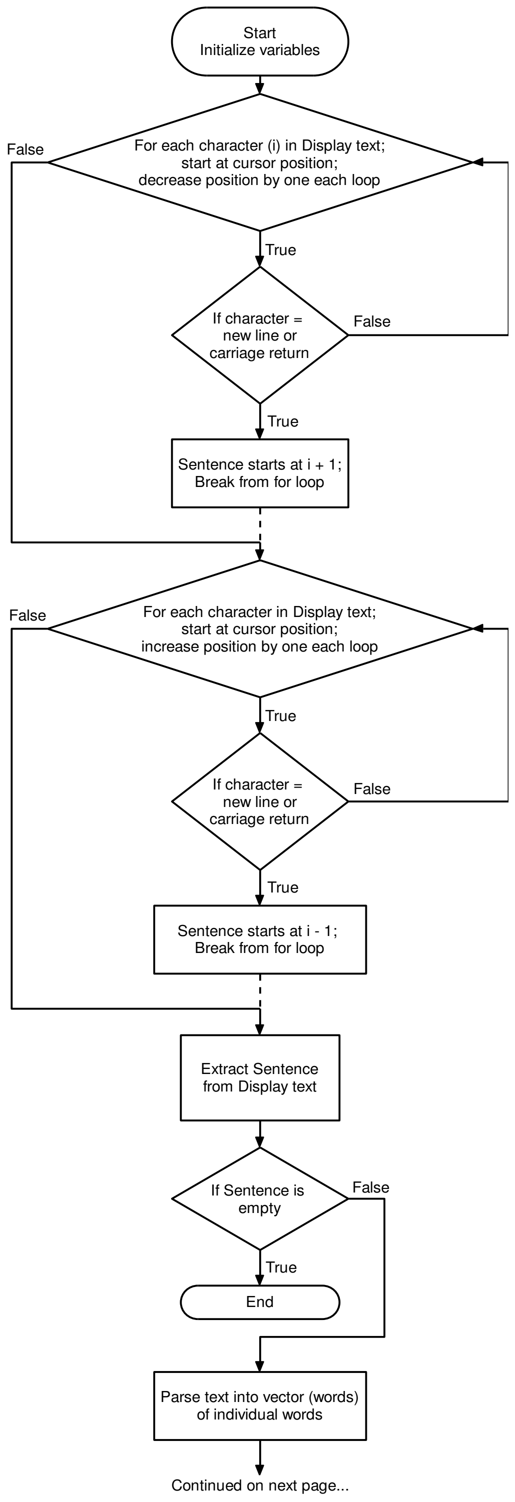
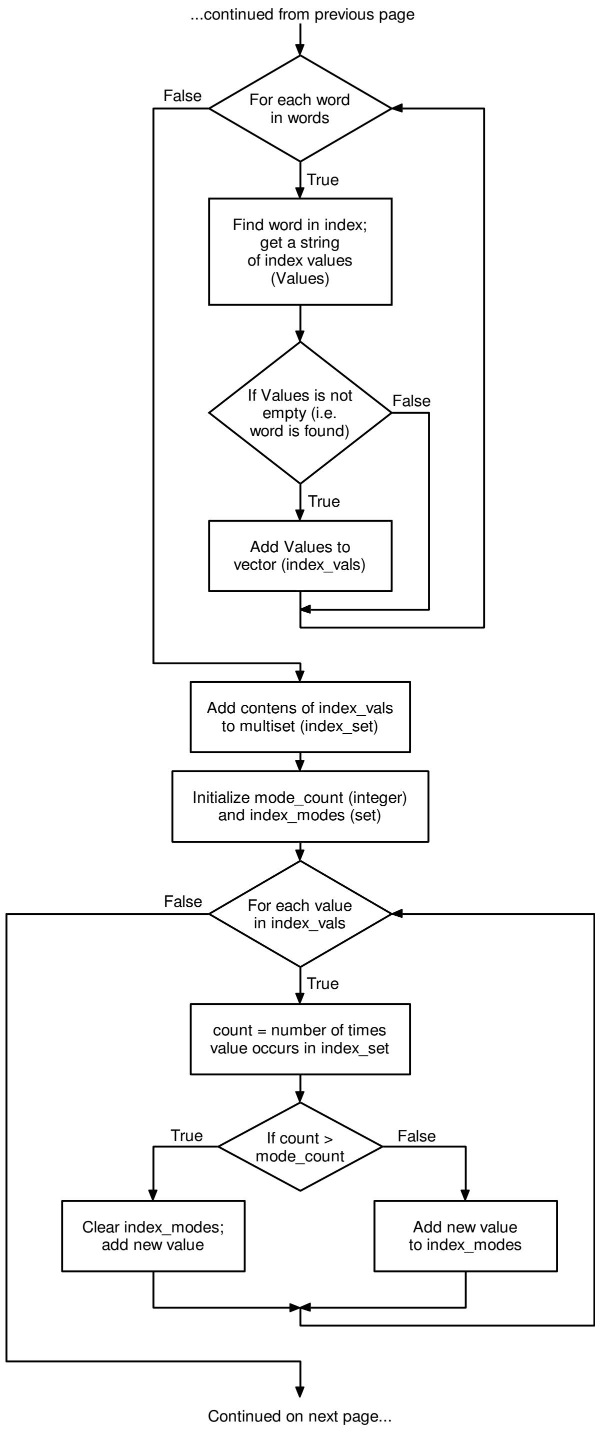
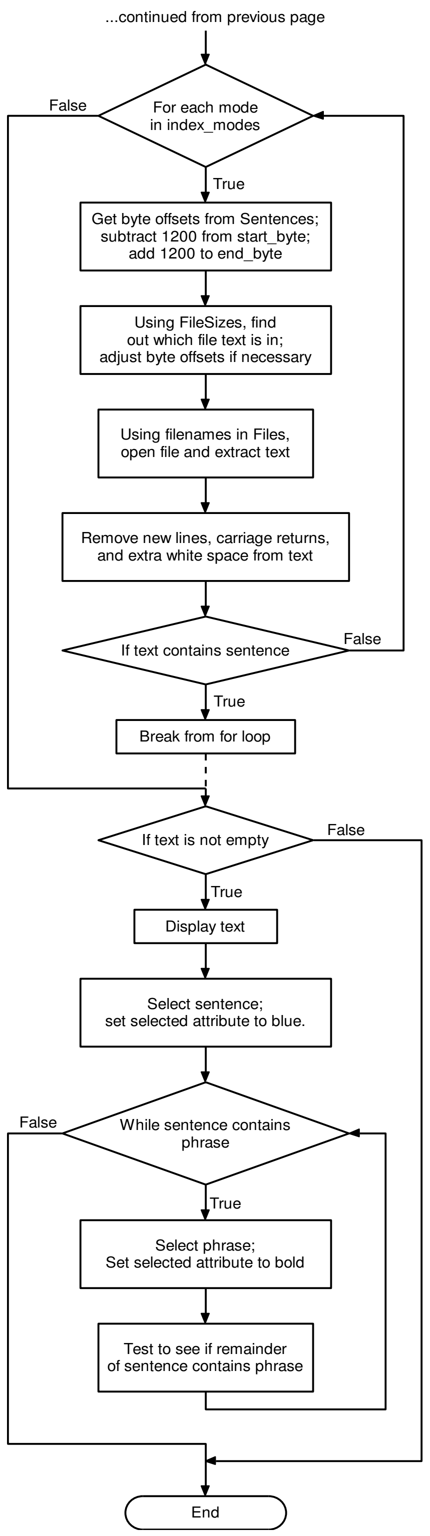

Start
Initialize variables
For each character (i) in Display text; start at cursor position; decrease position by one each loop
False
True
If character = new line or carriage return
False
True
Sentence starts at i + 1; Break from for loop
For each character in Display text; start at cursor position; increase position by one each loop
False
True
If character = new line or carriage return
False
True
Sentence starts at i - 1; Break from for loop
Extract Sentence from Display text
If Sentence is empty
False
True
End
Parse text into vector (words) of individual words
Continued on next page...

Figure B.8: Flow chart for GetParagraph function (1 of 3)



...continued from previous page
For each word in words
False
True
Find word in index; get a string of index values (Values)
If Values is not empty (i.e. word is found)
False
True
Add Values to vector (index_vals)
Add contens of index_vals to multiset (index_set)
Initialize mode_count (integer) and index_modes (set)
For each value in index_vals
False
True
count = number of times value occurs in index_set
If count > mode_count
True
False
Clear index_modes; add new value
Add new value to index_modes
Continued on next page...

Figure B.9: Flow chart for GetParagraph function (2 of 3)



...continued from previous page
For each mode in index_modes
False
True
Get byte offsets from Sentences; subtract 1200 from start_byte; add 1200 to end_byte
Using FileSizes, find out which file text is in; adjust byte offsets if necessary
Using filenames in Files, open file and extract text
Remove new lines, carriage returns, and extra white space from text
If text contains sentence
False
True
Break from for loop
If text is not empty
False
True
Display text
Select sentence; set selected attribute to blue.
While sentence contains phrase
False
True
Select phrase; Set selected attribute to bold
Test to see if remainder of sentence contains phrase
End

Figure B.10: Flow chart for GetParagraph function (3 of 3)

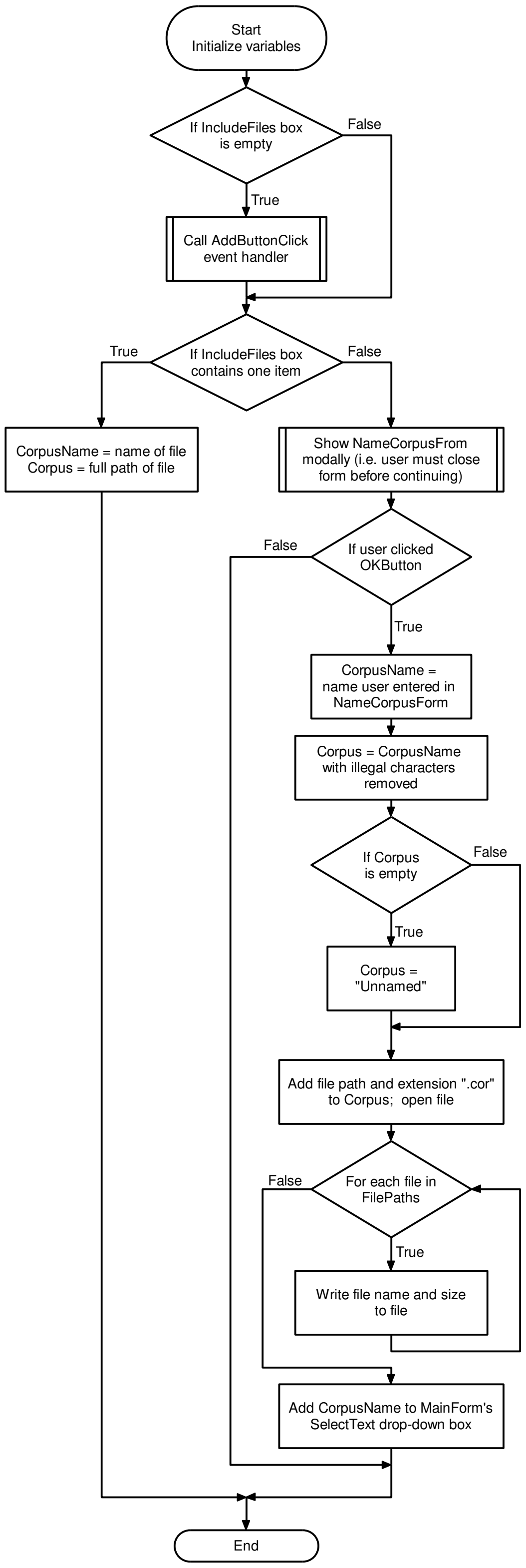

Start
Initialize variables
If IncludeFiles box is empty
False
True
Call AddButtonClick event handler
If IncludeFiles box contains one item
True
False
CorpusName = name of file
Corpus = full path of file
Show NameCorpusFrom modally (i.e. user must close form before continuing)
If user clicked OKButton
False
True
CorpusName = name user entered in NameCorpusForm
Corpus = CorpusName with illegal characters removed
If Corpus is empty
False
True
Corpus = "Unnamed"
Add file path and extension ".cor" to Corpus; open file
For each file in FilePaths
False
True
Write file name and size to file
Add CorpusName to MainForm's SelectText drop-down box
End

Figure B.11: Flow chart for TDirectory::OKButtonClick event handler

Appendix C

Programming Code

The C++ Builder IDE generates most of the code for visual components and provides many functions for common Windows tasks like displaying text and saving and opening files. The code included in this appendix is the code that I actually wrote: none of code generated by C++ Builder is included.

C.1 MainForm

C.1.1 Class declaration

```
class TMainForm : public TForm
{
private: // User declarations
   void LoadText(String);
   int Concord(String);
   void GetPhrases(std::vector<String>*, std::vector<int>*, String);
   int FormPhrases(std::vector<String>*, std::vector<int>*,
       std::vector<bool>*, std::vector<bool>*, String, bool, bool);
   void UpdateCollocates(std::vector<String>*, std::vector<int>*,
        String);
   void SortLists(std::vector<String>*, std::vector<int>*);
   bool Contains(std::vector<String>*, String);
   void DisplayPhrases(String);
   bool IsStopWord(String, int);
   void GetParagraph(int, String);
   void DisplayMessage(String);

   TStringList *Concordance;
   TStringList *PhrasesList;
   std::vector<String> *Phrases;
   std::vector<int> *PhrasesCount;
   std::vector<String> *IndexWords;
   std::vector<String> *IndexValues;
   std::vector<int> *Sentences;
   std::vector<String> *Files;
   std::vector<int> *FileSizes;

   int INDEX;
   int PHRASE;
   clock_t click1;
   clock_t click2;
public: // User declarations
   TStringList *SelectTextPaths;
};
```

C.1.2 MainForm event handlers

```
// Form create event
__fastcall TMainForm::TMainForm(TComponent* Owner)
        : TForm(Owner)
{  //Constants
   INDEX = 1;
   PHRASE = 2;

   SelectTextPaths = new TStringList;

   TStringList *Corpora = new TStringList;
   try
   {  Corpora->LoadFromFile("corpora.ini");
      for (int i = 0; i < Corpora->Count; i++)
      {  String CorpusName = Corpora->Names[i];
         SelectText->Items->Add(CorpusName);
         SelectTextPaths->Add(Corpora->Values[CorpusName]);
      }
      SelectText->ItemIndex = 0;
   }
   __finally
   {  delete Corpora;
   }

}
//--------------------------------------------------------------------
void __fastcall TMainForm::OKButtonClick(TObject *Sender)
```

```
{
   if (SelectText->Text == "")
   {  Application->MessageBox("Please select a text first", "",
                              MB_OK);
   }
   else
   {  if (!Files) LoadText(SelectTextPaths->Strings[0]);

      delete Concordance;
      delete PhrasesList;
      delete Phrases;
      delete PhrasesCount;

      Concordance = new TStringList;
      PhrasesList = new TStringList;

      Phrases = new std::vector<String>;
      PhrasesCount = new std::vector<int>;

      ListBox->Clear();
      Display->Clear();

      TCursor OriginalCursor = Screen->Cursor;
      Screen->Cursor = crHourGlass;
      ProgressForm->Caption = "Finding phrases";
      ProgressForm->Label->Caption = "Finding phrases with \'" +
                                     EnterWord->Text+ "\'...";
      ProgressForm->Gauge->MinValue = 0;
      ProgressForm->Gauge->Progress = 0;
      ProgressForm->Gauge->MaxValue = 20;
      ProgressForm->Visible = true;
      Application->ProcessMessages();

      int total_words = Concord(EnterWord->Text);

      if (total_words == 0)
      {  ProgressForm->Visible = false;
         if (IsStopWord(EnterWord->Text.LowerCase(), INDEX))
         {  DisplayMessage(EnterWord->Text);
         }
         else
         {  String Temp = "The word \"" + EnterWord->Text + "\"
                          was not found";
```

```
            char *message = Temp.c_str();
            Application->MessageBox(message, "Word not found",
                                    MB_OK);
            StatusBar->SimpleText = "";
         }
      }
      else if (total_words > 0)
      {  ProgressForm->Gauge->Progress =
            ProgressForm->Gauge->Progress + 1;
         //Estimate of how long search will take
         ProgressForm->Gauge->MaxValue = total_words / 4;

         GetPhrases(Phrases, PhrasesCount, EnterWord->Text);
         ProgressForm->Visible = false;

         if (Phrases->size() > 0) SortLists(Phrases, PhrasesCount);
         else Application->MessageBox("No expressions found.", "",
                                       MB_OK);

         Phrases->push_back(EnterWord->Text);

         for (int i = 0; i < Phrases->size() - 1; i ++)
         {  PhrasesList->Append(Phrases->at(i) + "  (" +
                                 PhrasesCount->at(i) + ")");
         }
         PhrasesList->Append("View all occurrences of \"" +
            EnterWord->Text + "\" (" + total_words + ")");
         ListBox->Items = PhrasesList;
         ListBox->ItemIndex = 0;
         DisplayPhrases(Phrases->at(0));
         StatusBar->SimpleText = "Double click on any sentence to
                                   see more.";

      }
   Screen->Cursor = OriginalCursor;
   }
}
//---------------------------------------------------------------------
void __fastcall TMainForm::FormClose(TObject *Sender,
                                     TCloseAction &Action)
{  TStringList *Settings = new TStringList;
   Settings->Values["StopWordsMsg"] =
      String(int(StopWordsMsg->DoNotShowThis->Checked));
```

```
Settings->Values["Articles"] =
   String(int(StopWordsForm->Articles->Checked));
Settings->Values["Conjunctions"] =
   String(int(StopWordsForm->Conjunctions->Checked));
Settings->Values["SubjectPronouns"] =
   String(int(StopWordsForm->SubjectPronouns->Checked));
Settings->Values["ObjectPronouns"] =
   String(int(StopWordsForm->ObjectPronouns->Checked));
Settings->Values["RelativePronouns"] =
   String(int(StopWordsForm->RelativePronouns->Checked));
Settings->Values["PossessiveAdjectives"] =
   String(int(StopWordsForm->PossessiveAdjectives->Checked));
Settings->Values["Numbers"] =
   String(int(StopWordsForm->Numbers->Checked));
Settings->Values["Quantifiers"] =
   String(int(StopWordsForm->Quantifiers->Checked));
Settings->Values["Demonstratives"] =
   String(int(StopWordsForm->Demonstratives->Checked));
Settings->Values["Be"] =
   String(int(StopWordsForm->Be->Checked));
Settings->Values["Have"] =
   String(int(StopWordsForm->Have->Checked));
Settings->Values["Do"] =
   String(int(StopWordsForm->Do->Checked));
Settings->Values["ModalVerbs"] =
   String(int(StopWordsForm->ModalVerbs->Checked));
Settings->Values["Not"] =
   String(int(StopWordsForm->Not->Checked));
Settings->Values["To"] =
   String(int(StopWordsForm->To->Checked));
Settings->SaveToFile(Directory->ProgramDirectory +
                     "\\settings.ini");
delete Settings;

TStringList *Corpora = new TStringList;
for (int i = 0; i < SelectText->Items->Count; i++)
{  Corpora->Values[SelectText->Items->Strings[i]] =
      SelectTextPaths->Strings[i];
}
Corpora->SaveToFile(Directory->ProgramDirectory +
                    "\\corpora.ini");
delete Corpora;
delete SelectTextPaths;
```

```
   delete Concordance;
   delete Phrases;
   delete PhrasesCount;
   delete PhrasesList;
   delete Directory->FilePaths;
   delete StopWordsForm->StopList;
   delete IndexWords;
   delete IndexValues;
   delete Sentences;
   delete Files;
   delete FileSizes;
}
//---------------------------------------------------------------------
void __fastcall TMainForm::ListBoxClick(TObject *Sender)
{
   int item = ListBox->ItemIndex;
   DisplayPhrases(Phrases->at(item));

}
//---------------------------------------------------------------------
void __fastcall TMainForm::ExitClick(TObject *Sender)
{
   Close();
}
//---------------------------------------------------------------------
void __fastcall TMainForm::FileLoadClick(TObject *Sender)
{
   int result = Directory->ShowModal();
      if (result == mrOk && Directory->Corpus != "")
      {  delete IndexWords;
         delete IndexValues;
         delete Sentences;
         LoadText(Directory->Corpus);
      }
}
//---------------------------------------------------------------------
void __fastcall TMainForm::SelectTextChange(TObject *Sender)
{
   delete IndexWords;
   delete IndexValues;
   delete Sentences;
   LoadText(SelectTextPaths->Strings[SelectText->ItemIndex]);
}
```

```
//---------------------------------------------------------------------
void __fastcall TMainForm::DisplayMouseUp(TObject *Sender,
      TMouseButton Button, TShiftState Shift, int X, int Y)
{  if (Display->Text != "")
   {  if (click1 <= 0 && Button == mbLeft) click1 = clock();
      else
      {  click2 = clock();
         int time = click2 - click1;
         if (time < 300)
         {  GetParagraph(Display->SelStart,
                         Phrases->at(ListBox->ItemIndex));
         }
         click1 = 0;
      }
   }

}
//---------------------------------------------------------------------
void __fastcall TMainForm::StopWordsClick(TObject *Sender)
{
   StopWordsForm->ShowModal();
}
//---------------------------------------------------------------------
void __fastcall TMainForm::SavePhrasesClick(TObject *Sender)
{
SaveDialog->FileName = EnterWord->Text;
if (SaveDialog->Execute())
{
   if (SaveDialog->FileName != "")
   {  ListBox->Items->SaveToFile(SaveDialog->FileName);
   }
}
}
//---------------------------------------------------------------------
void __fastcall TMainForm::SaveSentencesClick(TObject *Sender)
{
Viewer->SaveDialog->FileName = Phrases->at(ListBox->ItemIndex);
if (Viewer->SaveDialog->Execute())
{
   if (Viewer->SaveDialog->FileName != "")
   {  if (Viewer->SaveDialog->FilterIndex == 1)
      {  Display->PlainText = false;
         Viewer->SaveDialog->DefaultExt = "rtf";
      }
      else
      {  Display->PlainText = true;
         Viewer->SaveDialog->DefaultExt = "txt";
      }
      Display->Lines->SaveToFile(Viewer->SaveDialog->FileName);
   }
}
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::RemoveTagsClick(TObject *Sender)
{
   RemoveTags->Checked = !RemoveTags->Checked;
}
//---------------------------------------------------------------------------
void __fastcall TMainForm::AboutClick(TObject *Sender)
{
   AboutBox->ShowModal();
}
```

C.1.3 LoadText

```
void TMainForm::LoadText(String FileName)
{  TCursor OriginalCursor = Screen->Cursor;
   Screen->Cursor = crHourGlass;
   ProgressForm->Caption = "Loading Text";
   ProgressForm->Label->Caption = "Loading Text...";
   ProgressForm->Gauge->MinValue = 0;
   ProgressForm->Gauge->MaxValue = 200000;
   ProgressForm->Gauge->Progress = 0;
   ProgressForm->Visible = true;
   Application->ProcessMessages();

   IndexWords = new std::vector<String>;
   IndexValues = new std::vector<String>;
   Sentences = new std::vector<int>;
   Files = new std::vector<String>;
   FileSizes = new std::vector<int>;

   String WordsFileName = "";
   String ValuesFileName = "";
   String SenFileName = "";
   String Extension = "";
   String Path = "";

   for (int i = FileName.Length(); i > 0; i--)
   {  if (FileName[i] == '.')
      {  if (i != FileName.Length())
         {  Extension = FileName.SubString(i, FileName.Length());
         }
         FileName = FileName.SubString(1, i-1);

      }
      else if (FileName[i] == '\\')
      {  Path = FileName.SubString(1, i);
         FileName = FileName.SubString(i+1, FileName.Length());
         break;
      }
   }
   if (Path == "") Path = Directory->ProgramDirectory + "\\";

   WordsFileName = FileName + ".wdx";
   ValuesFileName = FileName + ".vlx";
   SenFileName = FileName + ".sen";

   if (Extension == ".cor")
   {  String File = Path + FileName + Extension;
      std::ifstream corpus_file(File.c_str());
         while (!corpus_file.eof())
         {  char buffer[1000];
            corpus_file.getline(buffer, 1000);
            String Line = String(buffer);
            for (int i = Line.Length(); i > 0; i--)
            {  if (Line[i] == '?')
               {  Files->push_back(Line.SubString(1, i-1));
                  FileSizes->push_back(StrToInt(Line.SubString
                     (i+1, Line.Length() - i)));
                  break;
               }
            }
         }
   }
   else
   {  int handle;
      String File = Path + FileName + Extension;
      handle = open(File.c_str(), O_BINARY);
      int file_size = filelength(handle);
      Files->push_back(File);
      FileSizes->push_back(file_size);
   }

   String WordsFile = Path + WordsFileName;
   String ValuesFile = Path + ValuesFileName;
   String SenFile = Path + SenFileName;
   std::ifstream read_words(WordsFile.c_str());
   std::ifstream read_values(ValuesFile.c_str());
   std::ifstream read_sen(SenFile.c_str());

   if (!read_words.fail() && !read_values.fail() &&
       !read_sen.fail())
   {
      {  while(!read_sen.eof())
         {  ProgressForm->Gauge->Progress =
               ProgressForm->Gauge->Progress + 1;
            char buffer[20];
            read_sen.getline(buffer, 20);
            Sentences->push_back(String(buffer).ToInt());
         }
         ProgressForm->Gauge->MaxValue =
            ProgressForm->Gauge->Progress * 2.5;
         while(!read_values.eof())
         {  ProgressForm->Gauge->Progress =
               ProgressForm->Gauge->Progress + 1;
            char buffer[1000000];
            read_values.getline(buffer, 1000000);
            IndexValues->push_back(String(buffer));
         }
         while(!read_words.eof())
         {  ProgressForm->Gauge->Progress =
               ProgressForm->Gauge->Progress + 1;
            char buffer[100];
            read_words.getline(buffer, 100);
            IndexWords->push_back(String(buffer));
         }
      }
   }
   else
   {  char* Msg = "Phraser needs to create an index for the text(s)
```

```
             that you have selected.  An index allows the
             program to locate words in the text(s) very
             quickly.  This may take several minutes,
             depending on the size of the text(s).  Would you
             like to continue?";
int result = Application->MessageBox(Msg, "Create index",
             MB_OKCANCEL);

if (result == ID_OK)
{
IndexWords->push_back("!");
IndexValues->push_back("0");
Sentences->push_back(-1);

String Buffer = "";
for (int i = 0; i < Files->size(); i++)
   {  std::ifstream file(Files->at(i).c_str(), ios::binary);
      int file_size = FileSizes->at(i) + 1;
      char *buffer = new char[file_size];
      try
      {  file.read(buffer, FileSizes->at(i));
         Buffer = Buffer + buffer;
      }
      __finally
      {  delete buffer;
      }
   }

ProgressForm->Gauge->MaxValue = Buffer.Length();

String Word = "";
String Tag = "";
String SenTemp = "";
bool in_tag = false;
int sen_num = 0;
for (int i = 1; i <= Buffer.Length(); i++)
   {  ProgressForm->Gauge->Progress =
         ProgressForm->Gauge->Progress + 1;
      if (i == 45980)
      {  int wsde = 0;
      }

      if (Buffer[i] == '<')
```

```
{  in_tag = true;
}
if (in_tag) Tag = Tag + Buffer[i];
else
{  if ((Buffer[i] == '\'' && Word.Length() > 0) ||
      (Buffer[i] >= 65 && Buffer[i] <= 90) ||
      (Buffer[i] >= 97 && Buffer[i] <= 122))
   //Tests to see if Buffer[i] is an apostrophe or a
   //letter (capital or lowercase)
   {  Word = Word + Buffer[i];
      SenTemp = SenTemp + Buffer[i];
   }
   else if (Word == "");
   else if (IsStopWord(Word.LowerCase(), INDEX))
   {  Word = "";
      SenTemp = SenTemp + Buffer[i];
   }
   else
   {  //Find word in Index
      Word = Word.LowerCase();
      if (Word[Word.Length()] == '\'')
      {  Word = Word.SubString(1, Word.Length() - 1);
      }
      int size = IndexWords->size();
      int low = 0;
      int hi = size;
      int mid = size / 2;
      std::vector<String>::iterator MiddleWords =
         IndexWords->begin() + mid;
      std::vector<String>::iterator MiddleValues =
         IndexValues->begin() + mid;

      while (hi - low > 1)
      {  if (*MiddleWords == Word)
         {  *MiddleValues = *MiddleValues + "," +
                              sen_num;
            hi = low;
         }
         else if (*MiddleWords > Word)
         {  hi = mid;
            MiddleWords += (hi + low) / 2 - mid;
            MiddleValues += (hi + low) / 2 - mid;
            mid = (hi + low) / 2;
```

```
               }
               else
               {  low = mid;
                  MiddleWords += (hi + low) / 2 - mid;
                  MiddleValues += (hi + low) / 2 - mid;
                  mid = (hi + low) / 2;
               }
            }
            if (*MiddleWords < Word)
            {  IndexWords->insert(++MiddleWords, Word);
               IndexValues->insert(++MiddleValues, sen_num);
            }
            else if (*MiddleWords > Word)
            {  IndexWords->insert(MiddleWords, Word);
               IndexValues->insert(MiddleValues, sen_num);
            }
         Word = "";
         SenTemp = SenTemp + Buffer[i];
         }
      }
      if (!in_tag && (Buffer[i] == '.' || Buffer[i] == '!' ||
          Buffer[i] == '?') || Tag == "</s>" || Tag == "<s/>"
          || Tag == "</p>" || Tag == "<p/>" || Tag == "</u>"
          || Tag == "<u/>")
      {  char qwetert = Buffer[i];
         char asdff = Buffer[i-1];
         Sentences->push_back(i-1);
         sen_num++;
         if (SenTemp.Pos("lunatic") > 0)
         {  int rty = 5;
         }
         SenTemp = "";
      }
      if (Buffer[i] == '>')
      {  in_tag = false;
         Tag = "";
      }
   }
WordsFile = Directory->ProgramDirectory + "\\" +
            WordsFileName;
ValuesFile = Directory->ProgramDirectory + "\\" +
             ValuesFileName;
std::ofstream words_file(WordsFile.c_str());
```

```
      std::ofstream values_file(ValuesFile.c_str());
      std::vector<String>::iterator IterWords =
          IndexWords->begin() + 1;
      std::vector<String>::iterator IterValues =
          IndexValues->begin() + 1;
      words_file.write(IterWords->c_str(), IterWords->Length());
      values_file.write(IterValues->c_str(), IterValues->Length());
      IterWords++;
      IterValues++;
      while (IterWords != IndexWords->end())
      {  words_file << "\n";
         words_file.write(IterWords->c_str(), IterWords->Length());
         values_file << "\n";
         values_file.write(IterValues->c_str(),
             IterValues->Length());
         IterWords++;
         IterValues++;
      }

      SenFile = Directory->ProgramDirectory + "\\" + SenFileName;
      std::ofstream sen_file(SenFile.c_str());
      std::vector<int>::iterator IterSen = Sentences->begin();
      String Temp = IntToStr(*IterSen);
      sen_file.write(Temp.c_str(), Temp.Length());
      IterSen++;
      while (IterSen != Sentences->end())
      {  sen_file << "\n";
         Temp = IntToStr(*IterSen);
         sen_file.write(Temp.c_str(), Temp.Length());
         IterSen++;
      }
   }
   }
   Screen->Cursor = OriginalCursor;
   ProgressForm->Visible = false;
}
```

C.1.4 Concord

```
int TMainForm::Concord(String Word)
{  Word = Word.LowerCase();
   String Values = "";
```

```
int size = IndexWords->size();
int low = 0;
int hi = size;
int mid = size / 2;
std::vector<String>::iterator MiddleWords =
   IndexWords->begin() + mid;
std::vector<String>::iterator MiddleValues =
   IndexValues->begin() + mid;

while (hi - low > 1)
{  if (*MiddleWords == Word)
   {  Values = *MiddleValues;
      hi = low;
   }
   else if (*MiddleWords > Word)
   {  hi = mid;
      MiddleWords += (hi + low) / 2 - mid;
      MiddleValues += (hi + low) / 2 - mid;
      mid = (hi + low) / 2;
   }
   else
   {  low = mid;
      MiddleWords += (hi + low) / 2 - mid;
      MiddleValues += (hi + low) / 2 - mid;
      mid = (hi + low) / 2;
   }
}
if (Values == "") return 0;
else
{  std::vector<int> index_vals;
   String TempVal = "";
   String PrevVal = "";

   for (int i = 1; i <= Values.Length(); i++)
   {  if (Values[i] != ',') TempVal = TempVal + Values[i];
      else
      {  if (TempVal != PrevVal)
         {  index_vals.push_back(StrToInt(TempVal));
         }
         PrevVal = TempVal;
         TempVal = "";
      }
```

```
    }
    index_vals.push_back(StrToInt(TempVal));

    int file = 0;
    int start_from = 0;
    std::ifstream text_file;
    text_file.open(Files->at(file).c_str(), ios::binary);

    for (int i = 0; i < index_vals.size(); i++)
    {  int start_byte = Sentences->at(index_vals[i]) + 1 -
                         start_from;
       int end_byte = Sentences->at(index_vals[i] + 1) -
                       start_from;
       while (end_byte > FileSizes->at(file))
       {  start_from = start_from + FileSizes->at(file);
          start_byte = start_byte - FileSizes->at(file);
          end_byte = end_byte - FileSizes->at(file);
          file++;
          if (file < FileSizes->size())
          {  text_file.close();
             text_file.open(Files->at(file).c_str(), ios::binary);
          }
       }
       text_file.seekg(start_byte, ios::beg);

       String Sentence = " ";
       char ch;
       for (int j = start_byte; j <= end_byte; j++)
       {  text_file.get(ch);
          if (ch == '\n')
          {  if (Sentence[Sentence.Length()] != ' ')
             {  Sentence = Sentence + " ";
             }
          }
          else if (ch != '\r' && ch != '\t' && !(ch == ' ' &&
                   Sentence[Sentence.Length()] == ' ' ))
          {  Sentence = Sentence + ch;
          }
       }
       Concordance->Append(Sentence.Trim());
    }
 text_file.close();
 return index_vals.size();
```

```
   }
}
```

C.1.5 GetPhrases

```
void TMainForm::GetPhrases(std::vector<String>* Phrases,
     std::vector<int>* PhrasesCount, String Headword)
{  std::vector<String>* TwoWords = new std::vector<String>;
   std::vector<int>* TwoWordsCount = new std::vector<int>;
   std::vector<bool>* TwoWordsStopLeft = new std::vector<bool>;
   std::vector<bool>* TwoWordsStopRight = new std::vector<bool>;

   std::vector<String>* ThreeWords = new std::vector<String>;
   std::vector<int>* ThreeWordsCount = new std::vector<int>;
   std::vector<bool>* ThreeWordsStopLeft = new std::vector<bool>;
   std::vector<bool>* ThreeWordsStopRight = new std::vector<bool>;

   std::vector<String>* FourWords = new std::vector<String>;
   std::vector<int>* FourWordsCount = new std::vector<int>;
   std::vector<bool>* FourWordsStopLeft = new std::vector<bool>;
   std::vector<bool>* FourWordsStopRight = new std::vector<bool>;

   int cut_off = 3;
   ProgressForm->Gauge->Progress =
      ProgressForm->Gauge->Progress + 1;
   FormPhrases(TwoWords, TwoWordsCount, TwoWordsStopLeft,
               TwoWordsStopRight, Headword, false, false);

   for (int i = 0; i < TwoWords->size(); i++)
   {  ProgressForm->Gauge->Progress =
         ProgressForm->Gauge->Progress + 1;
      int num = FormPhrases(ThreeWords, ThreeWordsCount,
                ThreeWordsStopLeft, ThreeWordsStopRight,
                TwoWords->at(i), TwoWordsStopLeft->at(i),
                TwoWordsStopRight->at(i));
      if (TwoWordsCount->at(i) - num < cut_off)
      {  (*TwoWordsStopLeft)[i] = true;
      }
   }

   for (int i = 0; i < ThreeWords->size(); i++)
   {  ProgressForm->Gauge->Progress =
            ProgressForm->Gauge->Progress + 1;
        int num = FormPhrases(FourWords, FourWordsCount,
                    FourWordsStopLeft, FourWordsStopRight,
                    ThreeWords->at(i), ThreeWordsStopLeft->at(i),
                    ThreeWordsStopRight->at(i));
        if (ThreeWordsCount->at(i) - num < cut_off)
        {  (*ThreeWordsStopLeft)[i] = true;
        }
    }

    for (int i = 0; i < TwoWords->size(); i++)
    {  if (!TwoWordsStopLeft->at(i) && !TwoWordsStopRight->at(i))
        {  Phrases->push_back(TwoWords->at(i));
           PhrasesCount->push_back(TwoWordsCount->at(i));
        }
    }

    for (int i = 0; i < ThreeWords->size(); i++)
    {  if (!ThreeWordsStopLeft->at(i) && !ThreeWordsStopRight->at(i)
            && !Contains(Phrases, ThreeWords->at(i)))
        {  Phrases->push_back(ThreeWords->at(i));
           PhrasesCount->push_back(ThreeWordsCount->at(i));
        }
    }

    for (int i = 0; i < FourWords->size(); i++)
    {  if (!FourWordsStopLeft->at(i) && !FourWordsStopRight->at(i)
            && !Contains(Phrases, FourWords->at(i)))
        {  Phrases->push_back(FourWords->at(i));
           PhrasesCount->push_back(FourWordsCount->at(i));
        }
    }

    delete TwoWords;
    delete TwoWordsCount;
    delete TwoWordsStopLeft;
    delete TwoWordsStopRight;

    delete ThreeWords;
    delete ThreeWordsCount;
    delete ThreeWordsStopLeft;
    delete ThreeWordsStopRight;
```

```
   delete FourWords;
   delete FourWordsCount;
   delete FourWordsStopLeft;
   delete FourWordsStopRight;

}
```

C.1.6 Contains

```
bool TMainForm::Contains(std::vector<String>* List, String Word)
{
   for (int i = 0; i < List->size(); i++)
   {  if (List->at(i).LowerCase() == Word.LowerCase()) return true;
   }
   return false;
}
```

C.1.7 FormPhrases

```
int TMainForm::FormPhrases(std::vector<String>* Phrases,
    std::vector<int>* PhrasesCount, std::vector<bool>*
    PhrasesStopLeft, std::vector<bool>*, PhrasesStopRight,
    String Headword, bool is_stop_left, bool is_stop_right)
{  std::vector<String>* LeftCollocate = new std::vector<String>;
   std::vector<int>* LeftCount = new std::vector<int>;;
   std::vector<String>* RightCollocate = new std::vector<String>;
   std::vector<int>* RightCount = new std::vector<int>;

   String HeadwordLC = Headword.LowerCase();

   int cut_off = 3;  //The cut off point for the frequency of
                     //the collocates.

   for (int i = 0; i < Concordance->Count; i++)
   {  String Sentence = Concordance->Strings[i];
      String SentenceLC = Sentence.LowerCase();

      int position = SentenceLC.Pos(HeadwordLC);

      while(position > 0)
      {  bool found_word = true;
```

```
String LeftWord = "";
String RightWord = "";

if (position != 1)
{  char test_letter = SentenceLC[position - 1];
   if (test_letter >= 97 && test_letter <= 122)
   {  found_word = false;
   }
}

int position2 = position + HeadwordLC.Length();
if (position2 < SentenceLC.Length())
{  char test_letter = SentenceLC[position2];
   if (test_letter >= 97 && test_letter <= 122)
   {  found_word = false;
   }
   if (test_letter == '\''
        && position2 < SentenceLC.Length())
   {  char test_letter2 = SentenceLC[position2 + 1];
      if (test_letter2 >= 97 &&  test_letter2 <= 122)
      {  found_word = false;
      }
   }
}

if(found_word)
{  //Find left collocate
   for (int j = position - 2; j > 0; j--)
   {  if (SentenceLC[j] == '\'')
      {  if (j > 1)
         {  if (SentenceLC[j-1] < 97 ||
                 Sentence[j-1] > 122)
            {  LeftWord = SentenceLC.SubString(j + 1,
                           position - 2 - j);
               break;
            }
         }
      }
      else if (SentenceLC[j] < 97  || SentenceLC[j] > 122)
      {  LeftWord = SentenceLC.SubString(j + 1,
                     position - 2 - j).Trim();
         break;
      }
```

```
            else if (j == 1)
            {  LeftWord = SentenceLC.SubString(1, position - 2);
            }
         }

         //Find right collocate
         for (int j = position2 + 1;
              j <= SentenceLC.Length(); j++)
         {  if (SentenceLC[j] == '\'')
            {  if (SentenceLC[j+1] < 97 || Sentence[j+1] > 122)
               {  RightWord = SentenceLC.SubString(position2 + 1,
                                j - position2 - 1);
                  break;
               }
            }
            else if (SentenceLC[j] < 97 || SentenceLC[j] > 122)
            {  RightWord = SentenceLC.SubString(position2 + 1,
                             j - position2 - 1).Trim();
               break;
            }
         }
      }

      if (LeftWord != "")
      {  UpdateCollocates(LeftCollocate, LeftCount, LeftWord);
      }
      if (RightWord != "")
      {  UpdateCollocates(RightCollocate, RightCount, RightWord);
      }

      int position3 =  SentenceLC.SubString(position2,
          SentenceLC.Length() - position2).Pos(HeadwordLC) - 1;
      if (position3 > 0)
      {  position = position + position3 + HeadwordLC.Length();
      }
      else position = 0;
   }
}

int left_total = 0;
int right_total = 0;

for (int i = 0; i < LeftCollocate->size(); i++)
```

```
   {  if (LeftCount->at(i) >= cut_off)
      {  Phrases->push_back(LeftCollocate->at(i) + " " +
            Headword);
         PhrasesCount->push_back(LeftCount->at(i));
         //If the newly added left collocate is a stop word,
         //the value for PhrasesStopLeft is true.
         if (IsStopWord(LeftCollocate->at(i).LowerCase(), PHRASE)
             || (StopWordsForm->To->Checked &&
             LeftCollocate->at(i).LowerCase() == "to"))
         {  PhrasesStopLeft->push_back(true);
         }
         //Otherwise it is false
         else
         {  PhrasesStopLeft->push_back(false);
            left_total = left_total + LeftCount->at(i);
         }
         //The value for PhrasesStopRight is equivalent to the
         //previous phrase's StopRight
         PhrasesStopRight->push_back(is_stop_right);
      }
   }

for (int i = 0; i < RightCollocate->size(); i++)
   {  if (RightCount->at(i) >= cut_off)
      {  Phrases->push_back(Headword + " " +
            RightCollocate->at(i));
         PhrasesCount->push_back(RightCount->at(i));
         //If the newly added right collocate is a stop word
         //the value for PhrasesStopRight
         if (IsStopWord(RightCollocate->at(i).LowerCase(),
             PHRASE))
         {  PhrasesStopRight->push_back(true);
         }
         //Otherwise it is false
         else
         {  PhrasesStopRight->push_back(false);
            right_total = right_total + RightCount->at(i);
         }
         //The value for PhrasesStopLeft is equivalent to the
         //previous phrase's StopLeft
         PhrasesStopLeft->push_back(is_stop_left);
      }
   }
   delete LeftCollocate;
   delete LeftCount;
   delete RightCollocate;
   delete RightCount;

   if (right_total > left_total) return right_total;
   else return left_total;
}
```

C.1.8 UpdateCollocates

```
void TMainForm::UpdateCollocates(std::vector<String>* Collocate,
        std::vector<int>* Count, String Word)
{  for (int i = 0; i < Collocate->size(); i++)
      {  if (Collocate->at(i).LowerCase() == Word.LowerCase())
            {  (*Count)[i] = Count->at(i) + 1;
               return;
            }
      }
   Collocate->push_back(Word);
   Count->push_back(1);
}
```

C.1.9 IsStopWord

```
bool TMainForm::IsStopWord(String Word, int type)
{  if (type == INDEX)
   {  int stop_length = 46;
      String StopWords[] = {  //ARTICLES
                              "a",
                              "an",
                              "the",
                              //PREPOSITIONS
                              "at",
                              "by",
                              "for",
                              "from",
                              "in",
                              "of",
                              "on",
"to",
"with",
//CONJUNCTIONS
"and",
"as",
"but",
"or",
//DEMONSTRATIVES
"this",
"that",
"these",
"those",
//FORMS OF "BE"
"am",
"are",
"be",
"been",
"is",
"was",
"were",
//PRONOUNS
"he",
"her",
"him",
"i",
"it",
"me",
"she",
"them",
"they",
"us",
"we",
"you",
//SPOKEN UTERANCES
"uh",
"um",
"huh",
"oh",
"hum",
//MISCELLANEOUS
"not",
"there"};
```

```
      for (int i = 0; i < stop_length; i++)
      {  if (StopWords[i] == Word) return true;
      }
   }
   else if (type == PHRASE)
   {  std::vector<String> StopWords = *StopWordsForm->StopList;
      for (int i = 0; i < StopWords.size(); i++)
      {  if (StopWords[i] == Word) return true;
      }
   }
   return false;
}
```

C.1.10 DisplayMessage

```
void TMainForm::DisplayMessage(String Word)
{  Word = Word.LowerCase();

   String Temp;
   if (Word == "the" || Word == "a" || Word == "an")
   {  Temp = "The word \"" + Word + "\" is an article and is always
             used before nouns.  For help with using articles, click
             on Advanced -> StopWords... from the menu.  Uncheck the
             box next to \"Articles.\" Then, try searching for
             various nouns such as \"people\", \"time\" or
             \"work.\"";
   }
   else if (Word == "at" || Word == "by" || Word == "from" ||
            Word == "in" || Word == "of" || Word == "on" ||
            Word == "to" || Word == "with")
   { Temp = "The word \"" + Word + "\" is a preposition and is
            typically used before nouns and after verbs.
            For help with using prepositions, try searching for
            various nouns such as \"home\", \"time\", or \"work\",
            or for verbs like \"hold\", \"put\", or \"look\"";
   }
   else if (Word == "and" || Word == "as" || Word == "but" ||
            Word == "or")
   {  Temp = "The word \"" + Word + "\" is a conjunction and is
             used to join other words or phrases.  For help with
             conjunctions, click on Advanced -> StopWord... from the
             menu.  Uncheck the box next to \"Conjunctions.\"  Then,
          try searching for other types of words that are likely
          to be used with conjunctions.";
}
else if (Word == "this" || Word == "that" || Word == "these" ||
         Word == "those")
{  Temp = "The words \"this\", \"that\", \"these\", and \"those\"
           are called demonstratives, and are frequently used
           with nouns, as in \"this house\" or \"those people.\"
           \"That\" is also used as a conjunction, as in \"the
           house that I live in.\"  For help with using
           demonstratives, click on Advanced -> StopWord... from
           the menu.  Make sure the box next to \"Demonstratives
           \" is unchecked.  Then, try searching for various
           nouns like \"house\" and \"people.\"";
}
else if (Word == "am" || Word == "are" || Word == "be" ||
         Word == "been" || Word == "is" || Word == "was" ||
         Word == "were")
{  if (Word == "be") Temp = "The word \"be\" ";
   else Temp = "The word \"" + Word + "\" is a form of the word
                \"be.\"  \"Be\" ";
   Temp = Temp + "is one of the most common words in English.
          For help with using the various forms of \"be\", click
          on Advanced -> StopWord... from the menu.  Make sure
          the box next to \"Forms of 'be'\" is unchecked.  Then,
          try searching for other types of words that are likely
          to be used with \"be.\"";
}
else if (Word == "he" || Word == "her" || Word == "him" ||
         Word == "i" || Word == "me" || Word == "she" ||
         Word == "them" || Word == "they" || Word == "us" ||
         Word == "we" || Word == "you")
{  if (Word == "i") Word = "I";
   Temp = "The word \"" + Word + "\" is a pronoun.  Pronouns are
           generally used with verbs, as in \"she likes him.\"
           For help with using pronouns, click on Advanced ->
           StopWord... from the menu.  Uncheck the boxes next to
           \"Subject pronouns\"  and \"Object pronouns.\" Then
           try searching for various verbs such as \"make\",
           \"see\", and \"know.\"";
}
else if (Word == "uh" || Word == "um" || Word == "huh" ||
         Word == "hum" || Word == "oh")
   {  Temp = "Words like \"" + Word + "\" are commonly used in
              speech to indicate a pause, hesitation, etc.  They are
              not generally part of any expresions.";
   }
   else if (Word == "not")
   {  Temp = "The word \"not\" is used to make verbs negative, as in
              \"I do not like sushi.\"  For help with this word,
              click on Advanced -> StopWord... from the menu.  Make
              sure the box next to \"Negative marker\" is unchecked.
              Then try searching for various verbs such as \"make\",
              \"see\", and \"know.\"";
   }
   else if (Word == "there")
   {  Temp = "\"There\" is one of the most frequently used words in
              English.  It occurs in the expressions \"there is\"
              and \"there are\", as in \"there is a small
              registration fee.\"  It is also used as the opposite of
              \"here.\"  For help with using \"there\", try
              searching for other words that are commonly used with
              it.";
   }


   char *message = Temp.c_str();
   Application->MessageBox(message, "High frequency word", MB_OK);
}
```

C.1.11 SortLists

```
void TMainForm::SortLists(std::vector<String>* List,
                          std::vector<int>* Count)
{  int size = List->size() - 1;
   int val1;
   int val2;
   String Temp;

   while(size > 0)
   {  for (int i = 0; i < size; i++)
      {  val1 = Count->at(i);
         val2 = Count->at(i+1);
         if (val1 < val2)
         {  (*Count)[i] = val2;
```

```
            (*Count)[i+1] = val1;
            Temp = List->at(i);
            (*List)[i] = List->at(i+1);
            (*List)[i+1] = Temp;
         }
      }
      size--;
   }

}
```

C.1.12 DisplayPhrases

```
void TMainForm::DisplayPhrases(String Phrase)
{  Phrase = Phrase.LowerCase();
   Display->Clear();
   Display->ScrollBars = ssNone;
   int where_are_we = 0;

   for (int i = 0; i < Concordance->Count; i++)
   {  String Sentence = Concordance->Strings[i];

      if (RemoveTags->Checked)
      {  bool in_tag = false;
         String Sentence2 = "";
         for (int i = 1; i <= Sentence.Length(); i++)
         {  if (Sentence[i] == '<') in_tag = true;
            if (!in_tag) Sentence2 = Sentence2 + Sentence[i];
            if (Sentence[i] == '>') in_tag = false;
         }
         Sentence = Sentence2;
      }

      String CleanSen = Sentence.LowerCase();
      for (int i = 1; i < CleanSen.Length(); i++)
      {  if (CleanSen[i] ==  '-') CleanSen[i] = ' ';
      }

      int sen_len = CleanSen.Length();
      int phrase_len = Phrase.Length();
      int position = CleanSen.Pos(Phrase);
```

```
bool sen_displayed = false;

while(position > 0)
{  bool found_word = true;

   if (position != 1)
   {  char test_letter = CleanSen[position - 1];
      if (test_letter >= 97 && test_letter <= 122)
      {  found_word = false;
      }
   }

   if (position + phrase_len <= sen_len)
   {  char test_letter = CleanSen[position + phrase_len];
      if (test_letter >= 97 && test_letter <= 122)
      {  found_word = false;
      }
   }

   if(found_word)
   {  if (!sen_displayed)
      {  Display->Lines->Append(Sentence);
         sen_displayed = true;
      }
      Display->SelStart = where_are_we + position - 1;
      Display->SelLength = phrase_len;
      Display->SelAttributes->Style =
         Display->SelAttributes->Style << fsBold;
      Display->SelAttributes->Color = clBlue;
   }

   int position2 =  CleanSen.SubString(position + phrase_len,
                    sen_len - position).Pos(Phrase) - 1;
   if (position2 > 0)
   {  position = position + position2 + phrase_len;
   }
   else
   {  position = 0;
      if (sen_displayed)
      {  where_are_we = where_are_we + sen_len + 4;
         Display->Lines->Append("");
         sen_displayed = false;
      }
```

```
         }
      }
   }
   Display->SelStart = 0;
   Display->SelLength = 0;
   Display->ScrollBars = ssVertical;
}
```

C.1.13 GetParagraph

```
void TMainForm::GetParagraph(int cursor, String Phrase)
{  TCursor OriginalCursor = Screen->Cursor;
   Screen->Cursor = crHourGlass;

   //Get the text surrounding the position at which the user clicked
   String Sentence = Display->Text;
   Display->SelLength = 0;
   int sen_start = 1;
   int sen_end = Sentence.Length();

   for (int i = cursor; i > 0; i--)
   {  if (Sentence[i] == '\n' || Sentence[i] == '\r')
      {  sen_start = i + 1;
         break;
      }
   }

   for (int i = cursor + 1; i <= Sentence.Length(); i++)
   {  if (Sentence[i] == '\r' || Sentence[i] == '\n')
      {  sen_end = i - 1;
         break;
      }
   }

   Sentence = Sentence.SubString(sen_start, sen_end -
              sen_start + 1).Trim();
   if (Sentence == "") return;

   std::vector<String> words;
   std::vector<int> index_vals;

   //Parse that text into a vector of individual words
```

```
String Word = "";
for (int i = 1; i <= Sentence.Length(); i++)
{  char letter = Sentence[i];
   if ((letter < 65 && letter != '\'') ||
      (letter > 90 && letter < 97) || letter > 122)
   {  if (Word != "")
      {  words.push_back(Word.LowerCase());
         Word = "";
      }
   }
   else Word = Word + letter;
}

//Look those words up in the index and make a vector of the
//index values
int gfdagffc = words.size();
for (int i = 0; i < words.size(); i++)
{  String Word = words[i].LowerCase();
   String Values = "";

   int size = IndexWords->size();
   int low = 0;
   int hi = size;
   int mid = size / 2;
   std::vector<String>::iterator MiddleWords =
      IndexWords->begin() + mid;
   std::vector<String>::iterator MiddleValues =
      IndexValues->begin() + mid;

   while (hi - low > 1)
   {  if (*MiddleWords == Word)
      {  Values = *MiddleValues;
         hi = low;
      }
      else if (*MiddleWords > Word)
      {  hi = mid;
         MiddleWords += (hi + low) / 2 - mid;
         MiddleValues += (hi + low) / 2 - mid;
         mid = (hi + low) / 2;
      }
      else
      {  low = mid;
         MiddleWords += (hi + low) / 2 - mid;
```

```
            MiddleValues += (hi + low) / 2 - mid;
            mid = (hi + low) / 2;
        }
    }
    if (Values != "")
    {  String TempVal = "";
       String PrevVal = "";

       for (int i = 1; i <= Values.Length(); i++)
       {  if (Values[i] != ',') TempVal = TempVal + Values[i];
          else
          {  if (TempVal != PrevVal)
             {  index_vals.push_back(StrToInt(TempVal));
             }
             PrevVal = TempVal;
             TempVal = "";
          }
       }
       index_vals.push_back(StrToInt(TempVal));
    }
}

//Fine the most frequent index values (the modes)
std::multiset<int> index_set;
for (int i = 0; i < index_vals.size(); i++)
{  index_set.insert(index_vals[i]);
}

int mode_count = -1;
std::set<int> index_modes;

for (int i = 0; i < index_vals.size(); i++)
{  int temp = index_vals[i];
   int count = index_set.count(index_vals[i]);
   if (count > mode_count)
   {  mode_count = count;
      index_modes.clear();
      index_modes.insert(index_vals[i]);
   }
   else if (count == mode_count)
   {  index_modes.insert(index_vals[i]);
   }
}
```

```
std::set<int>::iterator iter;
String FileText = "";
String FileTextNoReturn = " ";
int sen_pos = 0;

for (iter = index_modes.begin(); iter != index_modes.end();
     iter++)
{  //Get byte offsets of sentence, with 2400 bytes of
   //surrounding text
   int start_byte = Sentences->at(*iter) - 1200;
   if (start_byte < 0) start_byte = 0;
   int end_byte = Sentences->at(*iter + 1) + 1200;

   //Find out which file the text is in and adjust bytes
   //offsets if necessary
   int file = 0;
   while (start_byte > FileSizes->at(file))
   {  start_byte = start_byte - FileSizes->at(file);
      end_byte = end_byte - FileSizes->at(file);
      file++;
   }
   if (end_byte > FileSizes->at(file)) end_byte =
       FileSizes->at(file);

   //Get that text from the corpus
   std::ifstream text_file(Files->at(file).c_str(), ios::binary);
   text_file.seekg(start_byte, ios::beg);
   char ch;
   for (int i = start_byte; i <= end_byte; i++)
   {  text_file.get(ch);
      FileText = FileText + ch;
   }

   //Remove new lines, cariage returns and extra white space
   //from the text
   for (int i = 1; i <= FileText.Length(); i++)
   {  if (FileText[i] == '\n')
      {  if (FileTextNoReturn[FileTextNoReturn.Length()] != ' ')
         {  FileTextNoReturn = FileTextNoReturn + " ";
         }
      }
      else if (FileText[i] != '\r' && FileText[i] != '\t' &&
            !(FileText[i] == ' ' &&
            FileTextNoReturn[FileTextNoReturn.Length()] == ' ' ))
      {  FileTextNoReturn = FileTextNoReturn + FileText[i];
      }
   }
   //Verfiy that it is the correct text, and if so break
   sen_pos = FileTextNoReturn.Pos(Sentence);
   if (sen_pos > 0) break;
   else
   {  FileText = "";
      FileTextNoReturn = "";
   }
}

//Display the text, make the sentence blue, and make the
//phrase bold
if (FileTextNoReturn != " ")
{  Viewer->Display->Text = FileTextNoReturn;
   Viewer->Display->SelStart = sen_pos - 1;
   Viewer->Display->SelLength = Sentence.Length();
   Viewer->Display->SelAttributes->Color = clBlue;

   Phrase = Phrase.LowerCase();
   String SentenceLC = Sentence.LowerCase();
   int phrase_pos = SentenceLC.Pos(Phrase);
   int phrase_len = Phrase.Length();
   int sen_len = SentenceLC.Length();

   while(phrase_pos > 0)
   {  bool found_word = true;

      if (phrase_pos != 1)
      {  char test_letter = SentenceLC[phrase_pos - 1];
         if (test_letter >= 97 && test_letter <= 122)
         {  found_word = false;
         }
      }

      if (phrase_pos + phrase_len <= sen_len)
      {  char test_letter = SentenceLC[phrase_pos + phrase_len];
         if (test_letter >= 97 && test_letter <= 122)
         {  found_word = false;
         }
         }

         if(found_word)
         {  Viewer->Display->SelStart = sen_pos + phrase_pos - 2;
            Viewer->Display->SelLength = phrase_len;
            Viewer->Display->SelAttributes->Style =
               Display->SelAttributes->Style << fsBold;
         }

         int phrase_pos2 =  SentenceLC.SubString(phrase_pos +
            phrase_len, sen_len - phrase_pos).Pos(Phrase) - 1;
         if (phrase_pos2 > 0)
         {  phrase_pos = phrase_pos + phrase_pos2 + phrase_len;
         }
         else phrase_pos = 0;
         }
      Viewer->Display->SelStart = 0;
      Viewer->Display->ScrollBars = ssVertical;
      Viewer->Visible = true;
      }
   Screen->Cursor = OriginalCursor;
}
```

C.2 Directory

C.2.1 Class declaration

```
class TDirectory : public TForm
{
public: // User declarations
   String Corpus;
   TStringList *FilePaths;
   String ProgramDirectory;
   __fastcall TDirectory(TComponent* Owner);
};
```

C.2.2 Directory event handlers

```
// Form create event
__fastcall TDirectory::TDirectory(TComponent* Owner)
   : TForm(Owner)
```

```
{
   ProgramDirectory = Directory->DirectoryListBox->Directory;
}
//---------------------------------------------------------------------------
void __fastcall TDirectory::DriveComboBoxChange(TObject *Sender)
{
   DirectoryListBox->Drive = DriveComboBox->Drive;
   FileListBox->Drive = DriveComboBox->Drive;
   FileListBox->Directory = DirectoryListBox->Directory;

}
//---------------------------------------------------------------------------
void __fastcall TDirectory::OKButtonClick(TObject *Sender)
{  int result = mrNone;
   if (IncludeFiles->Items->Count == 0) AddButtonClick(Sender);
   String CorpusName;
   if (IncludeFiles->Items->Count == 1)
   {  CorpusName = IncludeFiles->Items->Strings[0];
      Corpus = FilePaths->Strings[0];
   }
   else if (IncludeFiles->Items->Count > 1)
   {  do
      {  result = NameCorpusForm->ShowModal();
      }
      while (NameCorpusForm->Name->Text == "" && result == mrOk);

      if (result == mrOk)
      {CorpusName = NameCorpusForm->Name->Text;
      Corpus = "";
      //Filters out characters which are not allowed in file names
      for (int i = 1; i <= CorpusName.Length(); i++)
      {  if (CorpusName[i] != '\\' &&
             CorpusName[i] != '/' &&
             CorpusName[i] != ':' &&
             CorpusName[i] != '*' &&
             CorpusName[i] != '?' &&
             CorpusName[i] != '"' &&
             CorpusName[i] != '<' &&
             CorpusName[i] != '>' &&
             CorpusName[i] != '|')
         {  Corpus = Corpus + CorpusName[i];
         }
      }
      if (Corpus == "") Corpus = "Unnamed";

      Corpus = ProgramDirectory + "\\" + Corpus + ".cor";
      std::ofstream corpus_file(Corpus.c_str());

      for (int i = 0; i < FilePaths->Count; i++)
      {  int handle;
         handle = open(FilePaths->Strings[i].c_str(), O_BINARY);
         int file_size = filelength(handle);
         corpus_file.write(FilePaths->Strings[i].c_str(),
                           FilePaths->Strings[i].Length());
         corpus_file << "?";
         String FileSize = IntToStr(file_size);
         corpus_file.write(FileSize.c_str(), FileSize.Length());
         corpus_file << "\n";
      }
      }
   }
   if (MainForm->SelectText->Items->IndexOf(CorpusName) < 0)
   {  MainForm->SelectText->Items->Insert(0, CorpusName);
      MainForm->SelectText->ItemIndex = 0;
      MainForm->SelectTextPaths->Insert(0, Corpus);
   }
   if (result != mrCancel) ModalResult = mrOk;
}
//--------------------------------------------------------------------

void __fastcall TDirectory::AddButtonClick(TObject *Sender)
{
   for (int i = 0; i < FileListBox->Items->Count; i++)
   {  if (FileListBox->Selected[i])
      {  if (IncludeFiles->Items->IndexOf
               (FileListBox->Items->Strings[i]) < 0)
         {  IncludeFiles->Items->Append
               (FileListBox->Items->Strings[i]);
            FileListBox->ItemIndex = i;
            FilePaths->Append(FileListBox->FileName);
         }
         else Application->MessageBox("File already added",
                 FileListBox->Items->Strings[i].c_str(), MB_OK);
      }
   }
}
```

```
//-------------------------------------------------------------------

void __fastcall TDirectory::RemoveButtonClick(TObject *Sender)
{  TStringList *Files = new TStringList;
   try
   {  for (int i = 0; i < IncludeFiles->Items->Count; i++)
      {  if (!IncludeFiles->Selected[i])
         {  Files->Append(IncludeFiles->Items->Strings[i]);
         }
      }
      IncludeFiles->Items = Files;
   }
   __finally
   {  delete Files;
   }
}
//-------------------------------------------------------------------

void __fastcall TDirectory::CancelButtonClick(TObject *Sender)
{
   IncludeFiles->Clear();
}
//-------------------------------------------------------------------


void __fastcall TDirectory::FormShow(TObject *Sender)
{
   delete FilePaths;
   FilePaths = new TStringList;
   IncludeFiles->Clear();
}
//-------------------------------------------------------------------
```

Appendix D

Quiz

1. Fill in the blank with the best choice.

I was looking for something different in my career, so I decided to ________ advantage of the new opportunities.

A. make B. take C. get D. find

(Hint: try searching for "advantage" in one of the corpora.)

2. Fill in the blank with the best choice.

She had asked him several times to ______ rid of the old refrigerator.

A. make B. take C. get D. put

(Hint: try searching for "rid" in one of the corpora.)

3. Fill in the blank with the best choice.

This street is very similar _____ the one I used to live on.

A. than B. of C. for D. to

(Hint: try searching for "similar" in one of the corpora.)

4. Fill in the blank with the best choice.

I can't find my camera. Can you help me look ______ it?

A. at B. like C. for D. up

(Hint: try searching for "look" in one of the corpora.)

5. Fill in the blank with the most likely word.

Did you wake ______ last night when the storm started?

(Hint: try searching for "wake" in one of the corpora.)

6a. Fill in the blank with the most likely word.

At the meeting, Mr. Smith spoke ______ behalf of the entire organization.

b. Is there another possible word that could be used? How can you tell which is more common?

(Hint: search for "behalf" in the Brown and Frown corpora.)

7a. Look up the word "apart" in the Switchboard corpus. How is this word normally used in this corpus (that is, in which phrase)?

b. Is this the only way you can use the word "apart"? How can you find out using the program?

8a. Look up the word "mind" in one of the corpora and look for phrases with the words "in", "to" and "on". (For example: "in mind", "on my mind"). Can you make any generalization about how each of these phrases is used?

b. Fill in the blank with the best choice (each choice will be used once).

I could not sleep well last night because I had a lot ______ mind.

I didn't have any definite plans _____ mind when I started the project.

A story my mother used to tell me always comes ______ mind when I visit Atlanta.

Choices: A. in B. to C. on my

9. Search for the word "keep" in several of the corpora. Try to find expressions that fit in the following sentences.

a. When writing a paper, it's important to ________________ who your audience is.

b. I had to run in order to ________________ them.

c. Can you ____________________ on the baby while I go upstairs?

d. If you work at home, it's important to ____________________ your business expenses.

Here are some possibilities (you will not use all of them): keep track of, keep a lid on, keep in, keep up with, keep in mind, keep pace with, keep on, keep an eye on.

10a. Suppose you search for the word "wrong" and notice that phrases like "what's wrong" and "what's wrong with" are very common. How might you find out if the phrase "what's right (with)" is also common?

b. Does the phrase appear to be common?

11. Search for the words "right" and "wrong" in one of the corpora. Try to find the phrases "go *or* went right" and "go *or* went wrong". Looking at the example sentences, do these phrases appear to be used in the same way? In other words, can you use "went right" in the same way that you can use "went wrong"? Explain why or why not.

12a. Search for the word “real” in the academic corpus and in Switchboard. Do you notice any differences in how the word is used in the phrases from each corpus? If so, what are they?
b. In what type of language, academic or conversational, are you more likely to find the phrase “real tired”?

13a. Search for the word “quarter” in the Switchboard and the Wall Street Journal corpus. In what type of language, conversational or business, are you more likely to hear “quarter”?
b. Does “quarter” seem to have a peculiar usage in that type of language? If so what?