April 28, 1994

SRC Research Report 123

Inside Hector: The Systems View

Loretta Guarino Reid and James R. Meehan

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Inside Hector: The Systems View

Loretta Guarino Reid and James R. Meehan

April 28, 1994

Affiliations
Loretta Guarino Reid is now at MicroUnity Systems Engineering Inc. (guarino@MicroUnity.com). James Meehan is now at Adobe Systems, Inc. (jmeehan@mv.us.adobe.com). Both authors worked at the Digital Equipment Corporation Systems Research Center during the course of the Hector project, October 1990–March 1993.

Abstract

Over a period of two and a half years, a team from the Systems Research Center designed, built, and revised a set of software tools for the dictionary division of Oxford University Press. Many aspects of this project were novel, including the approach to lexicography, the software environment, the problems of scale, and the demands of high performance and high reliability. In this paper, two members of the team describe some of the systems problems they faced in building these tools, and the solutions they devised to solve them.

Contents

1 Introduction

This paper describes a software project that operated within an unusual set of constraints. The software was built in a research laboratory and written in a language recently developed there, but it had to be reliable as soon as it went into use. The specifications constantly changed, yet we, the implementors, had to work within fairly strict time limits and ensure that program execution was fast at all times.

These constraints resulted in part from having users who were depending on the software for completing their own work within a specified time. In part, the nature of that work imposed its own limitations on us. We describe here the approaches we took to working within this unusual environment, and the compromises we had to make.

The project, called Hector, was a joint venture between the Systems Research Center of the Digital Equipment Corporation (SRC) and Oxford University Press (OUP). Hector was a feasibility study in computer-assisted corpus lexicography: its purpose was to create a database of information about words and their senses, based on evidence from a large corpus of English text drawn from a variety of sources. Indeed, each item in this database was to be directly linked to the corpus evidence on which it was based. The lexicographers from Oxford decided what information to put into the database; the computer scientists from SRC created the software tools that made it possible for the lexicographers to search the corpus, write the database entries, and create the links between them.

A mid-project report on Hector, published in Glassman et al.[1], describes these software tools. That report and an accompanying videotape were produced for an audience of lexicographers; they give the users' view of the Hector system. This report is intended for software engineers; it describes the implementors' view.

1.1 The Goal of the Project

From SRC's standpoint, the goal of the project was to explore the systems challenges in building software tools to enable lexicographers both to compile dictionaries and to extend their research in corpus lexicography.

SRC and OUP had agreed that the lexicographers would spend a year in Palo Alto, using the tools to write dictionary entries and to create the links from those entries to the corpus examples. At the end of the year, the lexicographers would return to Oxford and evaluate the project. As a result, we were under considerable time pressure. The tools needed to be ready when the lexicographers arrived, and they needed to continue to work at all times, so that the lexicographers could get a full year's work done to form the basis for the evaluation.

The most important challenge that we faced in building Hector was to keep the design flexible without completely sacrificing performance. Since we were exploring issues in the use of computers in lexicography, we expected the details of the specification to change over the course of the project. As the lexicographers tested their ideas about what would prove useful and why, they requested changes in the tools.

Performance was critical at all times. If the tools were too slow, the lexicographers would not use them. The tools also had to be robust. Lexicography is very labor-intensive; it was not acceptable to lose a lexicographer's work because of program errors or system problems.

1.2 Modula-3

Most of the Hector software was written in Modula-3[4], a strongly typed object-oriented programming language developed at SRC. We describe here briefly the features of the language that helped us work with the constraints presented in the previous section.

Modula-3's interfaces and strong type-checking made it much easier to make systematic design changes. We could feel confident that the compiler would flag all places that were affected by such changes. This feature was particularly important for Hector because the specifications were constantly changing. In addition, Modula-3's distinction between interfaces and implementations served us well when several of us were working on different parts of the same program.

Some features of Modula-3 made it much easier to design the structure of the Hector software. Objects and subtypes provided referential transparency and encapsulation, and were used extensively, especially in the graphics code. We made heavy use of threads, which are Modula-3's built-in support for concurrent processes. In addition, Modula-3's garbage collection relieved us of the design complexities of storage management.

The Modula-3 environment was still maturing at the start of the Hector project; there were no existing large applications in the language. We were early customers of many of the library and runtime facilities. As a result, we uncovered problems in threads, the scheduler, the garbage collector, and other parts of the system that affect all clients. Some Modula-3 facilities were rewritten several times to accommodate the size and speed requirements of Hector. However, the benefits provided by the Modula-3 language and environment offset these disadvantages for the project.

1.3 Terminology

In the rest of this report, we use some terminology that was specific to the Hector project. We present it here.

A *target word* is the word that is the object of a search of the on-line corpus.

A word can have one or more *senses*. For instance, the word "plaything" has one sense in which it is an object that is used as a toy. It also has a second sense, in which the word is used figuratively to describe a person who is the victim of another person or force. (He was the plaything of the gods.)

Each sense includes a *definition* and some grammatical information.

An *entry* is the collection of all the senses associated with a word, with some indication of the relationships between the senses. For instance, within an entry, all the noun senses might be grouped together, or the noun and verb senses of the same meaning might be grouped together, or the senses might be ordered by frequency.

When a lexicographer searches the on-line corpus for a certain word or pattern, the resulting list of all the occurrences of that word in its context is called a *concordance*. Each single example of the word in its context is called a *citation*.

Tagging a word in the corpus means linking it with some information in the database; in this paper, unless otherwise indicated, tagging will mean creating a link between a word in the corpus and a sense in an entry.

A *homograph* is a word of the same spelling as another but which has a different meaning. For example, "stock" in a financial context is a security; in a botanical context, a type of flower.

A *wordclass* is a part of speech.

Finally, a *collocate* is a word that typically occurs near another word. For example, "ham" is often collocated with "eggs", and "fish" with "chips".

1.4 Required features

From the outset of the project, the lexicographers required the software to provide certain basic functionality:

1. They wanted to test hypotheses about word senses by creating lexical search patterns and having the computer quickly find all the matching examples in the corpus. The patterns might include, not only a target word near other selected words, but also information about wordclass and membership in certain word lists we call lexical sets, described in section 3.5.

2. They needed to record their insights about word senses in a dictionary entry; as they understood more clearly the boundaries between word senses, they needed

to revise the structure of the entry, splitting one sense into two, combining several senses into a single broader sense, or making one sense a subsense of another.

3. They had to tag words in the corpus with their dictionary senses. The tools for searching the corpus and for composing the dictionary entry would therefore have to cooperate with each other.

1.5 Overview

This report explains the way in which we structured the software to provide the required functionality. We focus first on Argus, the corpus search tool, and second on Ajax, the dictionary entry editor. We examine, in particular, how the design of Argus and Ajax was affected by the following requirements: changing specifications, good performance, and robustness. Section 2 describes how Argus searched the corpus for examples of a word. In section 3 we look at how this feature was extended to permit more sophisticated searches. Section 4 shows how we attached senses to words in the corpus, and finally section 5 describes the tools for editing dictionary entries.

In appendix A, we evaluate how well the system described in the body of the report enabled us to meet the goals set out in the Hector overview [1]. We are not lexicographers, so we have not set our results in the context of related work by the lexicography research community.

Appendix B describes the details of a wordclass tagger for raw corpus text.

2 Searching the Corpus

The Oxford Hector Pilot Corpus contained 20 million words of running English text, with SGML[6] markings to indicate special characters and certain general classes of text, such as headings, signatures, and captions.

For various practical reasons, we decided that, after some initial cleanup work such as removing large duplicate sections, we would not change the text of the corpus during the course of the project, nor would we allow the lexicographers to edit the existing text.

As a consequence of this decision, we were able to simplify the design of the search tool. In order to search such a large corpus quickly for the patterns requested by the lexicographers, it was clear that we needed a full-text index. Since the corpus wasn't going to change, we simply used the index of each word in the corpus as a reliable way to identify it; word number 3,467,122 would not change during the project.

This decision also meant that we could afford to invest considerable time in precomputing lexical and syntactic information, which we added to the static index as well.

The tool that we built to search the text, and to view the corpus, was called Argus. In its simplest form, Argus resembled the Unix utility *grep*. However, we needed much better performance than *grep* provides. We also needed richer functionality, such as being able to search for more than one word at a time, and allowing various other constraints on the search.

The lexicographers needed to search the corpus and quickly extract all instances of the target word they were studying. They also needed to see the target words in a KWIC concordance. For example, the concordance for the word "stock" might include these citations:

```
at while Dai Ichi would change its stock portfolio this year, there wo
ought 500 April 2,150 puts. In the stock options, BP was the busiest,
 future." The 250,000 tonne buffer stock, which was bought in a vain a
ack enough to put it in order, and stock and equip it.  <hdl> Putting
 laborious, to carry water or move stock to it. When these top-priorit
```

The lexicographers also needed to sort a concordance in different ways. For instance, they might want to sort it by the words to the right of the target word, to uncover patterns of use.

The following subsections describe the parts of Argus and the supporting programs that locate and display concordances of a single word.

2.1 Parsing the Corpus

In order to search for words in the corpus, we needed to identify them by determining their boundaries. For instance, when is punctuation part of a word? In the sentence "I'd've drunk another pint", is "I'd've" one word or three?

To perform this lexical analysis, we used a parser from Houghton Mifflin. We ran the parser over each document in the corpus and stored the results in disk files, one for each corpus file. (We stored the corpus not as one large file, but as 276 separate files. This made the corpus slightly easier to manage early in the project when we were still modifying the text and the analysis tools.) We numbered all the words in the corpus sequentially from 1 to 20 million, across file boundaries, producing a unique index for each word.

The parser also performed syntactic analysis. It produced a set of four binary files containing wordclass tags, sentence boundaries, clause boundaries, and prepositional phrases. Each file contained a set of records; for example, a wordclass record contained the starting position of the word, its length, its wordclass tag, and its baseform. The Adam wordclass tagger, a separate program, produced a fifth file for each document.

We could not keep all 1,380 of these data files open all the time. Consequently, Argus spent a lot of time opening and closing files. Caching file handles and the static data from the files helped somewhat, but we believe that this disk activity reduced performance noticeably.

2.2 Indexing

The basic operation of searching for a word needed to be very fast. We implemented it with a precomputed index that mapped words to their positions in the corpus. Most of the search operations were implemented most naturally in terms of word indexes. We found that it simplified the structure of the system to split the search code into two parts. One computed results in terms of word indexes; the other translated word indexes into a file name and the character position in that file.

For performance reasons, we wanted only a single copy of the search code executing: the index was quite large. However, we needed to provide service to several users simultaneously. The index was therefore managed by a pair of servers, shared by all the users. One server, the *Index Server*, took search requests and returned a list of word indexes. The other server, the *Corpus Position Server*, took requests containing word indexes, and returned lists containing filenames and positions within those files. Each server had a front end that handled multiple TCP connections and controlled access to the database managers, which were single-threaded programs. Both servers were designed and implemented by our colleague Mike Burrows.

2.2.1 The Index Server

The Index Server implemented the search facility, returning the list of words that satisfied a search request. We discuss two aspects of the Index Server: the scheme for compact data representation used for the index file, and the implementation of the search operations.

The central index file was a sequence of alphabetically sorted records, each containing a word and the list of word indexes where the word occurred.To save space, both the words and the indexes were encoded. Each word was represented by the number of leading characters it shared with its predecessor in the file, and the text of the unshared suffix.

For example, suppose a document contained only the four words “propel”, “propeller”, “propellers”, and “propels”. Instead of storing all 32 characters, the index would contain only 11 characters plus 4 single-byte numbers, arranged like this:

word	count	suffix
propel	0	propel
propeller	6	ler
propellers	9	s
propels	6	s

Since the list of word indexes was in increasing order, it was stored as a list of consecutive differences, so the list 100150, 100170, 100185, 100202 would be stored as 100150, 20, 15, 17. Each individual number was stored as a sequence of 7-bit digits; the 8th (high-order) bit indicated the end of the sequence.

We kept sub-indexes to help locate a word and its list of word-indexes quickly. A sub-index file contained one entry for every 4,000 bytes of data in the index file. Each entry in the sub-index consisted of a byte-offset in the central file and the word and word index at that offset. Binary search in the sub-index yielded a location close to a word/location pair, from which the index file was searched using the word and word-index information from the sub-index entry.

This compression scheme saved us a factor of three in space; each index entry in the Hector databases (words, wordclasses, and so on) took 1.33 bytes on average. If the data had not been compressed, each pair would have taken more than 4 bytes per entry, since the position values didn’t fit in 3 bytes. Accessing the disk file containing the index contributed substantially to the time needed to query the corpus, so the compression sped up queries in addition to saving disk space.

Even with this compression scheme, the Hector index file required about 130 megabytes of storage.

Although the index representation was compact, it still permitted the search operations to be executed quite quickly. To find all the examples of a word, the Index Server located the record for that word using binary search. Then it generated all the locations where the word occurred by unpacking the list of indexes stored with the word.

We discuss the ways in which the Index Server implemented more complicated searches in section 3.

2.2.2 The Corpus Position Server

The function of the Corpus Position Server was straightforward. However, its implementation was not.

Given the index of a word, the Corpus Position Server returned the name of the document in which that word occurred, and its starting position within that document. A simple implementation might be an array that mapped a word index to a document name and a character position, but we exploited characteristics of the data (e.g., most words are relatively short) to store it more compactly.

We used three arrays: one contained the character position of every thousandth word; one recorded the length of all the other words; and one contained the character position of the start of each document. We used binary search to search the first array for the nearest position that was not past the target, then added up the lengths in the second array between that position and the target position. This gave us a map from word index to character index. Then we used the third array to translate from the absolute character index to a location within a particular document file.

The first array, the one we searched with binary search, contained an entry with the index and position of every word in the corpus that was longer than 15 characters (i.e., whose length could not be represented in 4 bits), as well as an entry for every thousandth word in the corpus, regardless of length.

The second array, used to home in on the target, contained 4-bit numbers representing the length of each of the 20 million words. (If the word actually occurred in the first array, it had a dummy entry in the second array because we never needed to look at the entry in that array.)

The three arrays together fit in less than 10 megabytes. This was small enough that the server kept them entirely in main memory. This permitted fast access to the data, and hence a rapid translation from word index to file name and character position.

2.3 Displaying Concordances

Having described the support tools for performing searches on the corpus, we now explain how we implemented the code that took a request for a target word from the lexicographer and displayed the resulting concordance.

The early versions of Argus merely applied the operations in a straightforward, linear way: first a call to the Index Server to get a list of word indexes, then a call to the Corpus Position Server, and finally a call to the display routine for each concordance line. The display routine extracted a 96-character text segment from the corpus file, centered around the target word, and displayed it on the screen.

We felt that since the corpus search code was fast, this approach would perform adequately. However, it proved to be too slow: the lexicographers were spending a lot of time waiting for results. So was Argus. After Argus had sent off a request to one of the three servers, it simply waited for the result before taking the next step. This wasn't necessary; the servers were designed to run independently, yet Argus was using only one of them at a time.

We therefore decided to exploit Modula-3's support for multi-threaded programs and improve the performance by pipelining the information flowing to and from the servers. Argus was modified to maintain a pair of consumer/producer queues for each of the servers, and three background threads managed the communication.

What went into the pipeline was a user request, and what came out was a set of matches that satisfied that request.

Apart from coordinating the various processes, the only hard part of implementing the pipeline was handling interrupts cleanly. After seeing a few lines of the concordance, the lexicographers might realize that their query was incomplete, incorrect, or likely to produce vast quantities of data, so they would cancel the search. When they did so, it was important to clear the pipeline completely.

This pipeline greatly reduced the time it took for the lexicographers to see the results of a search. It permanently altered the way in which they used the system, giving new support to the adage that performance *is* functionality. With the new implementation, the lexicographers felt free to make a series of successively more refined searches, rather than doing the most general possible search and then scrolling through the results.

2.4 Sorting the Concordances

It was important for the lexicographers to call up the concordance for a word quickly; it was also important that they be able to sort it quickly in different ways to gain insights into the sense divisions of words. Sorting by the words to the right of the target word, for example, was helpful for distinguishing the various senses of phrasal verbs: run along, run in, run into, run off, run over a bicycle, run over your lines, run over an account. Sorting by the words to the left helped distinguish senses of nouns: a first edition, the morning edition, the paperback edition, a smaller edition (of). It was also useful to sort the examples by the order of the documents within the corpus, which was organized roughly by genre: for example, journalism, fiction, correspondence. Finally, the lexicographers needed to sort a concordance by the order of the senses that had already been assigned to the target words.

In order for us to sort the citation in all these different ways, our internal representation included not only the text of the citation, but also its location in the corpus, a description of the document in which it was found, a list of all the individual words in the citation, and information to facilitate its display on the screen. The words themselves were similarly represented by text, position, sense-tag, and so on.

The obvious advantage to this scheme was that the text of the target word and its document order were already stored in memory by the time the concordance was displayed, so it was easy to sort by those criteria.

The obvious disadvantage was that citations were fairly heavyweight, requiring much more memory than the text alone. However, we observed that the lexicographers frequently repeated a search with slightly different parameters, so that the same citations would often be retrieved. Therefore we used a simple cache for the target word objects rather than re-creating them every time. Initially, there was no mechanism for emptying the cache; we simply relied on virtual memory to hold it all.

Normally this worked out satisfactorily; the users would usually quit the program before the workstation's memory manager was overloaded. But this did cause problems when the system was used to generate a wide variety of examples of different words (for a paper that one of the lexicographers was writing, not something that occurred in the day-to-day work of writing senses). We considered various automatic memory management schemes, but each had its drawbacks, so we decided eventually to provide a simple button labeled "Empty the Cache" and leave the decision up to the user, who was, after all, in the best position to know when the contents of the cache were no longer useful.

There was one other interesting design issue related to sorting. Sorting by the words to the right or left of the target word required access to the text of those words. (We could easily identify them by calculating their word indexes relative to the target word.)

The problem was deciding when to construct the heavyweight objects representing all the surrounding words in the citation. If we had deferred the decision until we were asked to sort the citations, there would have been a long delay while we waited for the database server. Instead, we constructed them in the process of displaying the citation line; the request for the surrounding words was part of the fourth stage of the pipeline. In hindsight, it should have been an independent, fifth stage of the pipeline. Although the 4-stage pipeline was fast enough for the lexicographers, a fifth stage would have given us even greater speed, since the citation could then be displayed before this information was retrieved.

3 More Sophisticated Searches

In addition to being able to search for a single word in the corpus, Argus also provided a variety of more sophisticated searching techniques. The lexicographers could search for any of several alternative words ("stock" or "investment" or "bond"), or constrain the search to a word belonging to one or more wordclasses ("stock" used as a noun or as an adjective). The word could also be constrained to occur in the presence of a collocate, which could have its own constraints. In this section, we describe the ways in which the basic search mechanisms were extended to provide this flexibility.

3.1 Sets of Words

The most common use of alternatives was in searching for a word along with its inflections, case variations, and spelling variations. The lexicographers considered these to be different examples of the same word. For example, the concordance with case variations and inflections of "stock" contained these citations:

```
olls like oats to make a palatable stock feed. Rye flour does not rise
kind of change in the rules of the Stock  Exchange," said Charles, ‘th
UBLISHER WITH A SCHEDULE OF UNSOLD STOCK AT THE DUE DATE OF PAYMENT. T
ring holidays here. There’s a well stocked supermarket, shops and a ra
lies may be restricted to conserve stocks. If you need a new light bul
ty people. We are one of the great stocks of Europe. We are the people
```

We provided code to generate the inflections and variations automatically. Our implementation was largely brute-force; we wrote some simple inflection routines, a small set of exceptions to those routines, and a table of spelling variations. This implementation gave us a 90% solution, but occasionally generated spurious inflected forms, like "importanter" and "importantest". However, it didn't cost the Index Server any significant time to check for a word that didn't occur in the corpus, and the lexicographers found it quite useful to have the program generate inflections so easily.

3.2 Wordclass Constraints

The lexicographers wanted to specify more than just literal words (text) in their search patterns; they wanted to find words that had certain properties as well.

For example, as described in [1], we used two programs, Adam and the Houghton Mifflin parser, to generate wordclass tags for all the words in the corpus, drawn from a set of over 300 wordclasses. This information was stored in the database. The lexicographers identified 20 general wordclasses.[1] Argus reduced the 300 to these 20, and we made them available for the lexicographers to use as constraints in the search pattern.

The Index Server built indexes for the wordclasses by treating the names of the wordclasses as special "words". (We added an initial character, "@", to each name, so that is was not possible to confuse a wordclass name with a word in the corpus.) Then we created an index for these wordclass names, parallel to the index of the words in the corpus. For example, this index made it appear as if the word "@NOUN" occurred wherever the corpus contained a word that had been marked as a noun. The procedure for finding a wordclass was therefore the same as the procedure for finding a word.

To find all the occurrences of "stock" used as a noun, the Index Server could have generated the list of all the positions where the word "stock" occurred, and the (very, very lengthy) list of all the positions where a noun occurred, and then taken their intersection. In practice, it achieved the same result, without actually generating the long intermediate lists, by traversing the indexes in parallel; the indexes were sorted in ascending order. The sub-index scheme described in section 2.2.1 let the Index Server quickly skip over large sections of the index without even reading it from the disk.

3.3 Syntactic Information

The Houghton Mifflin Parser also generated syntactic information for sentences. For each word in the corpus, it produced the baseform of the word. It computed the position and length of each sentence, prepositional phrase, clause, subject, and predicate. For prepositional phrases, the parser also produced a code for the preposition that headed the phrase. This information was stored in much the same way as the wordclass information.

The lexicographer could see this data on the screen by selecting a citation from the concordance and asking for a detailed description. We hoped this data would also be useful in restricting a search by syntax, finding only those citations, for example, in which the word "stock" occurs as the subject of a clause. Unfortunately, the syntactic analysis was often unreliable, and since it didn't include all the information that the lexicographers wanted, we abandoned this feature of searching.

[1]The set included noun, proper noun, verb, adjective, adverb, degree adverb, preposition, personal pronoun, reflexive pronoun, determiner, number, ordinal, modal, auxiliary, possessive, infinitive marker, negative, conjunction, and "other". There was some overlap in the small set—"myself" counted as both a personal pronoun and as a reflexive pronoun.

3.4 Collocates

The lexicographers often needed to search for target words that occurred in the presence of one or more collocates. To do this, the lexicographer specified the distance between that word and the desired collocate, either exactly or within a certain range. So for instance, the lexicographer could ask for all occurrences of the target word "stock" where "exchange" is within three words to the right.

```
s securities market, Francis Yuen, stock exchange chief executive, say
 1987 stock market crash, when the stock and futures exchanges closed
ght former senior officials of the stock exchange, including former ch
```

The lexicographer could use the same search constraints on a collocate as on a target word: wordclass, inflections, etc. In fact, the lexicographer could specify that the collocate be *any* word that satisfied a particular set of constraints.[2] For instance, the lexicographer could ask for all occurrences of "stock" followed immediately by any preposition:

```
derwriters worldwide began to take stock of the disaster, estimates we
g authorities.  At some stage, the stock of colleges, counties and reg
he literature of recipes ('Clarify stock to the brightness of sherry')
, encased with caul and bound with stock) of pig's trotter, came crisp
a casino, it's becoming a laughing stock around the world and the way
```

The lexicographers could also specify a collocate of a collocate. For instance, suppose a lexicographer working on the word "neck" wishes to see all instances of the idiom "breathe down one's neck". The search for the idiom is most easily expressed using a collocate of a collocate. The target is "neck" and its inflections. The collocate of the target is "breathe" and its inflections, within ten words to the left of "neck". The collocate of the collocate is "down" within five words to the right of "breathe".

```
nnifer Capriati breathing down her neck. These are rich and exciting t
With the French breathing down its neck, Northumbrian will have a more
 attentiveness, breathing down his neck. Toby gave up. Too tired to do
ury breathed down the authorities' necks if they hung on to it," the W
n't breathe down the back of their necks. Though he was a slave-driver
 were breathing furiously down the necks of the Opposition Party, and
```

[2] It was possible to do this for the target word too, but this rarely proved useful.

If the lexicographers had used a simpler search, they might well have missed the last three examples, where "breathe down" is separated from "neck" by several intervening words, and where "breathe" is also separated from "down".

Collocates introduced a fundamental change in the search requests to the Index Server. It was no longer sufficient to specify properties that applied to a single word; it was necessary to specify a relationship between several words in a search. These *positional* constraints on the search requests contained two pieces of information:

- The two words that had to satisfy the constraint; for example, the target word and a collocate, or two collocates.
- The distance between these two words.

In practice, a positional constraint between two words was cast as a condition on one of the words; the condition specified another word in the search request and a range of word indexes, where a negative index meant to the left of the word and a positive index meant to the right of the word.

To make this concrete, let's consider how the positional constraints would be expressed in some of the examples above. In the first example, "stock" with "exchange" within three words to the right, the search conditions for "exchange" would contain a positional constraint (written as "stock 1, 3"). In the third example, "breathe down one's neck", the search conditions for "breathe" would contain a positional constraint (neck -10, -1) and the search conditions for "down" would contain a positional constraint (breathe 1, 5).

These positional constraints complicated the search strategy of the Index Server. The Server would simultaneously search for words that satisfied the non-positional constraints of the request. In our first example, it would search for all the instances of "stock" and all the instances of "exchange". Then for each possible combination of "stock" and "exchange", it would check that the positional constraints were satisfied before reporting a match.

Of course, the structure of the indexes permitted the Index Server to perform such searches much more quickly than this simple-minded description would imply. The Index server used the positional constraints to skip over sections of the corpus that could not contain a match. If the next word that matched "stock" was, say, a thousand words beyond the next word that matched "exchange", there was no way to produce any matches in that thousand-word range, and the server could skip over all intervening occurrences of "exchange".

This was particularly important for examples like our second, all occurrences of "stock" followed immediately by any preposition. There are many prepositions in the corpus, but few of them will follow immediately after "stock", and it would be very slow if we had to examine each preposition individually to confirm this.

3.5 Lexical Sets

Quite late in the project we added another way of classifying words in the corpus, by their membership in what we called *lexical sets.* These sets were derived very simply from a thesaurus that Macquarie Inc. made available to us. For instance, the lexical set called *Creatures* included such words as "dog", "cat", and "fish", along with their case variants and inflections.

The sets are lexical, not semantic. *Creatures* is not the set of terms that actually refer to animals, but just a list of words. For example, the phrases "hot dog", "the cat is out of the bag", and "three fish short of a lawnmower" (an actual example from the corpus) contain items from *Creatures* that do not denote animals. But even this simple tool proved to be quite useful to the lexicographers, and we would design the lexical sets systematically and thoroughly if we were starting over.

Because the lexical sets were experimental, Argus was structured to make it easy to change the number and composition of the lexical sets. A library routine returned the words in the lexical sets at runtime, based on the contents of a text file.

In principle, a lexical set is quite similar to a wordclass. It is a property or value of a word (even encoded with an "@" name like the wordclass names), and the Index Server searched for words with that property. The lexical sets were quite large, however; the program for generating all the members of a lexical set in the corpus occasionally failed because it could not handle such large sets in a single pass.

Lexical sets introduced another problem. The lexicographer could ask to see all occurrences of the word "taste" that occurred within five words of any word belonging to the lexical set *Food*:

```
 loaded with sugar to kill the taste.  He keeps two diaries, one for
ight fruit juices, because the taste is too strong and sharp.  'Our f
has revealed that Jeff's sauce tastes much better on chicken wings th
ou wish 1 green pepper salt to taste 2 lbs chicken wings, jointed and
e rose.  All of the wines were tasted blind by a panel of three &dash
```

The semantics of "within five words" had to be "within five words before the target word, or five words after the target word, but not including the target word itself". Otherwise, if the target word were treated as a member of the same lexical set as the collocate, the search would return every occurrence of the target word.

For instance, we might want to distinguish uses of “orange” as a color from uses of “orange” as a fruit. Since colors often occurred together (“his orange and yellow ball”), we might have searched for the word “orange” within five words of any word in the lexical set *Color*. However, since “orange” can be a color and is always within five words of itself, all occurrences would have matched our search, which would have defeated the purpose. We referred to this problem as “auto-collocation.” To prevent it, we required that a match never use the target word to satisfy a collocate constraint.

The implementation of the Index Server made auto-collocation difficult to prevent automatically. Instead, we solved the problem in Argus, not in the Index Server, by splitting a position-range that included the target position into its left-hand and right-hand ranges, explicitly excluding the target position. For instance, if the lexicographer requested a collocate within three words of the target, Argus sent the Index Server two separate queries; one for collocates on the left, and one for collocates on the right. Argus merged the resulting lists into a single concordance.

4 Connecting Words and Definitions

Hector was not just a system for viewing static corpus data. As the lexicographers examined words in context, they acquired new insights about their meanings. To preserve this information, the lexicographers needed a way of recording the connections between words in the corpus and definitions they had written.

In this section, we review the implementation of the Sense Server, which maintained these connections, and we look at the way in which the connections were used in searching the corpus. Understanding how the Sense Server worked requires some explanation about how we named the senses (section 4.1), and how we recorded connections that were not simple mappings of one word to one sense (section 4.2).

4.1 Naming Senses

In order to establish the connections between the words in the corpus and their senses, we needed ways to name the things being connected.

Argus already provided a way of naming corpus words, by making them the target words of queries. The Argus user interface provided a sense-tag field with each concordance line that the lexicographer used to associate a sense with the target word in the line.

In addition, we needed a way to name the senses in an entry. If the number of senses, and their order, were established once and never changed thereafter, we could have named them according to this ordering. For example, *“fence”, homograph 1,* *sense 3.1.* But the sense numbers were changing all the time as an entry evolved and the relationships became clearer.

In order to provide a reliable reference to a sense, Ajax, the dictionary-entry editor, assigned each sense a six-digit unique identifier (UID) when the sense was created. This UID never changed and remained permanently associated with the sense, no matter how the sense was renumbered. It provided an unambiguous name for the sense, and it was used to establish the connections to corpus words.

It would have been awkward and error-prone for the lexicographers to use UIDs directly, so Ajax let them choose nicknames (mnemonics) to represent the UID for each sense. The lexicographers were free to choose whatever mnemonics they found suggestive. The mnemonics were meaningful only within a single entry; the same mnemonic could be used in a different entry, and these would have nothing to do with each other. The lexicographers changed the mnemonics whenever they liked.

As an example, consider this excerpt containing some of the senses from the entry for "leap". Each sense or subsense contains a UID, a mnemonic assigned by the lexicographer, some grammatical information, and a definition.

```
1   uid=508119 mnemonic=JUMP
    vi; usu with adjunct-dir; (of a person or animal) to jump
    forcefully or pounce suddenly
1.1 uid=522775 mnemonic=RUSH
    vi; with adjunct; (in metaphorical or hyperbolical use)
    (especially of a person) to make a sudden rush to do
    something; to act precipitately or impulsively in some
    situation; to start eagerly or quickly
1.2 uid=508132 mnemonic=JUMPTRANS
    vt; (of a person or animal) to jump over (a barrier or gap);
    often in metaphorical use
2   uid=508122 mnemonic=PROGRESS
    vi; with adjunct-dir; to progress or advance
    dramatically or suddenly
3   uid=508118 mnemonic=JUMPNOUN
    nc; a forceful jump or pounce
```

Argus mapped the mnemonics to UIDs and used the UIDs internally for identifying senses. Later, we discuss how Ajax managed the mapping between the UIDs and the senses, and how it communicated relevant changes to Argus.

4.2 Sense-Tagging

We originally thought that sense-tagging the corpus would be a simple mapping: each word in the corpus would be associated with one sense in the dictionary. The lexicographers, on the other hand, knew that language was not that straightforward. For example, puns typically map to more than one sense, while strings of words such as the idiom "kick the bucket" map to one sense. Metaphorical uses are a distinct category. And sometimes the lexicographers just couldn't be sure of the sense from the context. They wanted to express that a word might be one sense or possibly another.

We therefore had to develop a sense-tag notation to capture the range of connections that the lexicographers wanted to make. We added "or" expressions for indicating ambiguity, and we defined a set of suffix tags to indicate unusual uses. There were six modifiers, each written as a suffix on a basic tag:

-a	used as an adjective
-m	used as a metaphor
-n	used as a noun
-p	used as a proper noun
-x	an exploitation (neither literal nor metaphorical)
-z	auxiliary word of a multi-word lexical item (e.g., part of a fixed phrase)

We also allowed a question mark at the end of a tag, to indicate that the lexicographer was uncertain about the tag.

Most tags were *simple*: one mnemonic, no suffix. These were stored in the database compactly and uniformly, using just the single UID. But because expressions in the sense-tag notation could be arbitrarily long, the non-simple tags, referred to as *complex*, required a separate representation and a separate file in the database.

As an example of a complex sense-tag, the word "leap" in the sentence:

```
The leap into the unknown of German monetary unity is being greeted
with a serenity which borders on complacency.
```

was tagged as

```
JUMPNOUN-m or PROGRESS-x
```

which means that the lexicographers thought this was either a metaphorical use of the `JUMPNOUN` sense (as in, say, "a jump of 10 meters") or it was an exploitation of the `PROGRESS` sense (as in, say, "making a great leap forward").

Argus provided support for tagging multi-word lexical items, such as idioms, compounds, and phrasal verbs. Each word in a multi-word item had to be tagged with the same tag, since the entire item was associated with a single sense. We could have had the lexicographers use the basic sense-tag mechanism to tag each word in turn, but that would have been laborious and error-prone.

Instead, we provided a multi-word mode. In this mode, the lexicographer selected a main word under which the multi-word item would be indexed, then selected the remaining words in the item as auxiliary words, and finally assigned a tag to the entire group. The tag was associated with the main word; the same tag but with the suffix "-z" was associated with each of the auxiliary words. The words did not need to be contiguous. For example, you could tag the phrasal verb "boot out" in the sentence "His father had booted him out after catching him with a girl in his room".

4.3 The Sense Server

The Sense Server was a program that managed the sense-tag database. It performed two functions: it ensured that changes were applied to the database in a robust way, without any loss of data, and it captured the richness of the connections expressed by the sense-tag notation.

Since several lexicographers could be simultaneously reading and writing the files where the sense tags were stored, the Sense Server ensured that changes were synchronized, and that requests for sense tags always yielded the most recent assignments.

In order to represent the richness of the sense-tag notation while making it easy for the Index Server to use the sense assignments, the information about sense assignments was represented in two files. One was a binary file, suitable for use by the Index Server. The other was a text file that contained additional information about words that had been assigned complex tag expressions, rather than just a single, unmodified sense tag.

The binary file contained the index of the word in the corpus, the UID of the sense, the ID of the lexicographer who made the assignment, and a flag that indicated whether there was an entry for this word in the text file.

If the word was assigned a single, unmodified sense, then there was no information in the text file; the sense-tag expression was just the sense UID. If the sense assignment was a complex expression, then the Sense Server stored the full tag expression in the text file. (This file was not encoded, since there was wide variation in the sense-tag expressions, and there were relatively few complex tag expressions.)

We had originally attempted to manage the sense tags directly in Argus, without a separate Sense Server, but we finally concluded that we could not implement a consistent view of a shared, mutable file using NFS. When one copy of Argus rewrote the sense-tag file to record new assignments, there was no way to tell whether other instances were still using the old version, and hence whether it was safe to delete it. We found ourselves either referencing non-existent files or squandering huge quantities of disk space on obsolete versions of the sense-tag file that were of no interest to anyone.

4.4 Searching and Sorting on Senses

Argus allowed the lexicographers to use the sense-tagging information to search the corpus. The search could either include or exclude all words with a given sense tag; for example, the lexicographers might want to ignore all items that had already been tagged, or they might want to see only the words with a particular sense tag, or the words that had been assigned a tag unrelated to the current entry (to see, say, multi-word tags or just wrong tags).

Most sense-tag search conditions were implemented by the Index Server directly. The sense-tag assignments were just properties of words. When the sense-tag search condition was equivalent to checking whether a word had a specified sense assigned, the Index Server handled it.

However, if the lexicographer wished to include other tagged words and exclude untagged words, or vice versa, the logic required was more than the Index Server was designed to handle. These search conditions were implemented by a downstream sense-tag filter in Argus.

The presence of sense tags also provided new opportunities for sorting the concordance. Of the many possible ways of sorting, the lexicographers found these three to be useful:

1. Sort by UID. While the UIDs themselves were not meaningful to the lexicographers, this was the fastest way of seeing which lines had been given the same sense tag.

2. Sort in alphabetical order by mnemonic. See section 4.1 for the relations between UIDs and mnemonics.

3. Sort by dictionary order. The citations were sorted first by the headword, then by the homograph, and finally by the sense number.

The lexicographers could constrain a search by sense tag only for the target word, not for a collocate. By the end of the project, there were enough sense-tagged words in the corpus that the lexicographers wished they also had had the ability to constrain a collocate by sense tag.

5 Creating Dictionary Entries

Our motivation in creating Ajax, the dictionary-entry editor, was to hide the details of the encoding of dictionary entries so the lexicographers could focus on the contents of the entries.

Because Ajax managed the encoding, we could ensure that the entries were always consistently represented and could be analyzed easily by the computer. For instance, we wrote programs that checked that terms used in the Register or Field values of a word sense were consistent with the dictionary editorial policy. If this information hadn't been consistently encoded in the dictionary entries, we couldn't have checked it.

The lexicographers experimented with the organization and structure of the dictionary entries throughout the year, attempting to find a form that would let them easily express the relationships between word senses. We also experimented with ways to display the dictionary entry information on the screen, so that the lexicographers could see the information they needed when they needed it.

In this section, we review the changing structure for the dictionary entries and our strategies for dealing with these changes. We explain how we stored the dictionary entries and provided back-ups and versions for robustness. Then we review the different views of the entry that Ajax implemented, and how the views related to one another. We describe some of the implementation difficulties in implementing the Reorder and Renumber commands. Finally, we describe how Ajax sent Argus information about the mapping between UIDs and dictionary senses.

5.1 The Structure of a Dictionary Entry

When we first set out to work with the lexicographers to define the formal grammar of a dictionary entry, we assumed that the information and structure of a dictionary entry were well understood. We thought that there was consensus on what information dictionary entries contained and how that information was structured. We believed we were merely recording the grammar of the entry formally so that we could program the editor correctly. Our assumptions about dictionary entries were wrong.

For every grammar we proposed, the lexicographers could envision an entry that didn't fit the grammar. Our requirements and theirs were apparently at odds. They felt our formal grammar was confining, and wanted to leave as much flexibility as possible. We needed an unambiguous grammar so we could analyze entries reliably.

We revised the grammar of a dictionary entry several times as the lexicographers refined their representation. In fact, we continued to make changes to the grammar until the end of the project. Our early struggles with the grammar definition led us to design Ajax so that the entry grammar was defined by a specification file. Changing the grammar of the dictionary entries required that we change only the specification file, not the program itself.

When the grammar changed, we needed to convert any existing dictionary entries so they conformed to this new specification. Unfortunately, we did not mechanize this conversion, partly because there were technical problems (nesting and unnesting of semantic information), but mostly because we didn't realize soon enough that we should have made this a priority. As a result, once the lexicographers had written more than a handful of entries, we permitted only upward-compatible changes to the grammar, so the existing entries automatically satisfied the new grammar.

This was an instance where our system shortcomings affected the project goals. The lexicographers requested several grammar changes which we could not make because of the restriction that the changes be upward compatible. This hindered their exploration of effective ways to present the relationship of information in a dictionary entry.

Here is an example of the textual representation of an entry. Its structure is explicitly encoded using SGML.

```
<entry done=TRUE>above board
<lex>
<vf>above-board</vf>
<sen uid=516356 tag=board>
<gr>comp</gr>
<def>openly acknowledged, without concealment or deceit</def>
<ex>&ellip. everything about the lease was legal and
    above board.
<clues>=</clues></ex></sen></lex></entry>
```

5.2 Managing Entry Files

We stored each entry in its own file. This simplified access to the entries; if we had stored several entries per file, we might have needed to permit several lexicographers to modify that file at the same time. The simpler scheme also let us use the file system directly for backup and file locking.

This scheme worked well, although having lots of small files meant that we had to do some file management. Rather than keep all the files in one directory, we created a set of directories and used the first two letters of each headword to decide where to put each entry. As it turned out, for the number of entries completed during the year, we could have just used a flat directory structure for the new entries, but a complete dictionary would need the more elaborate strategy.

To locate the file containing a headword, we used an Ingres database. The database contained two tables: the dictionary table, which contained the mapping from headwords to filenames, and the UID table, which contained the mapping from UIDs to filenames. We planned to use the UID table for generating consistent cross-references, but this feature was never used by the lexicographers.

Using a database for headword-to-filename mapping was overkill, of course. We used Ingres because we originally envisioned using it to handle information such as wordclasses. When we changed those plans, we never got around to reimplementing Ajax to use a simpler scheme.

We used RCS[5], a standard Unix utility, to do backups and version management. When a lexicographer saved an entry, Ajax used RCS commands to write a version-history.

RCS does not export a library that can be called from a program, so we had to use the shell commands intended for human users. Ajax checked out the file, stored the text representation of the entry in the file, then checked the file back in, leaving a read-only version available. If any part of this commit operation failed, Ajax released the RCS lock on the file.

Because there was no RCS library, Ajax had to deal with forking processes and ensuring that it provided the right input under all circumstances. An error in this part of the implementation would have lost the lexicographers' work just as they thought they were finished.

RCS protected us against the unlikely event that two lexicographers were trying to save the same file at the same time. We did not use RCS to prevent two lexicographers from trying to edit the same entry at the same time. It seemed unnecessary because no two lexicographers would ever work on the same entry at the same time. We could have locked an entry by checking out the file the first time a lexicographer modified it.

5.3 Presenting Entries Effectively

The most difficult challenge in Ajax was how to present the dictionary entries effectively. The lexicographers wanted to see as much of the entry on the screen as possible, so they could get an overview of how the entry was developing. They also needed to change the entry easily, both the information in the fields of the entry and the structure of homographs, senses and subsenses. At the same time, we needed the separate fields of information cleanly marked, so we could encode the entry properly.

We provided several different solutions to this problem by presenting three different views of an entry. Each had a different set of virtues and shortcomings, and none proved ideal, but by switching between them judiciously, the lexicographers managed to get their work done. The three views were:

- The Complete Structure View: This view allowed all the information in an entry to be edited. It faithfully reflected the complete structure of the dictionary entry.
- The Set-of-Senses View[3]: This view concentrated on the individual senses: it included the mnemonic, homograph and sense numbers, grammar, definition, examples, register (formal, slang, etc.), cross-reference, and a general-purpose "note" field. This was the view that the lexicographers edited while they examined the corpus and decided how to break the word into senses and subsenses.
- The Print View: This was a read-only view that displayed the entry in a separate window, formatted as it would be for the printed dictionary. This provided the lexicographers with a compact, familiar format that made it easy to review their work or scan another entry quickly.

The Complete Structure View was a fairly direct representation of the hierarchical structure of the entry itself. The structure of dictionary entries specified the order of the fields and whether those fields were optional. For example, in any given word sense the (optional) inflections preceded the (optional) variant forms, which in turn preceded the (required) definition, which preceded the (required) examples. The lexicographers determined this order, which reflected the order in which the fields appeared in a printed dictionary entry.

In order to make as much of an entry visible on the screen as we could, our strategy was to display the fields in order, but to pack the fields together as tightly as possible, left to right, then top to bottom. We assumed that most fields had a typical width, which was given in the specification file. Ajax determined the size of a field by creating a box of the width in the specification file, then making the box as tall as necessary to display all the information in the field.

Our emphasis on compactness meant that we sacrificed spatial consistency, since each sense potentially had a different set of fields, and the fields were always packed

[3]This view was called a skeletal view in [1].

as tightly as possible. For example, the grammar field in an idiom sense appeared on the second line, while the same field for a non-idiom sense appeared on the first line.

One possible solution to the problem of spatial consistency would have been to set the order for the fields so that the mandatory fields always preceded optional ones, and thus always appeared in the same place. This would have meant keeping two different field orders in the specification file, since the current design asserted that the order of the fields was the same as the order in the printed dictionary entry.

The Set-of-Senses View presented a scrollable view with horizontal lines separating individual senses. The format of senses in this view was fixed; it was not driven from a specification file. This decision was based on schedule pressures: if we had had more time, we would have made this view specification-driven as well.

The Set-of-Senses View did not handle the full complexity of the entry structure; it supported only those fields that appeared in most senses. It didn't include information tied to a homograph rather than a sense (pronunciation, etymology, variant forms, etc.), nor did it allow, say, a pronunciation or an inflection that was specific to a sense. To create such complex entries, the lexicographer had to use the complete structure view.

The Print View of an entry was implemented by the Sid program, written at OUP. Sid is a general-purpose formatting program for text marked in SGML. We wrote a specification file describing the Hector output format, and we ran Sid in a separate process. We converted the entry to its text representation and piped the result to the Sid process.

5.4 Reordering and Renumbering

Early versions of Ajax managed all sense and homograph numbering automatically, based on the position of the sense or homograph in the entry hierarchy. As described in [1], we changed our strategy so that it was always the responsibility of the lexicographer to provide sense numbers. To help the lexicographers manage sense numbering, both editable views provided reordering and renumbering operations.

Implementing these operations proved more difficult than you might imagine at first glance. Sometimes the lexicographers wanted the senses to be rearranged to correspond to the sense numbers that they had assigned. At other times, they wanted the opposite: they wanted the sense numbers to be recomputed to correspond with the way they had arranged the senses. We satisfied both requirements. Rearranging the senses to reflect the sense numbers was called *reordering*. Recomputing the sense numbers based on the order of the senses was called *renumbering*. Either operation could be invoked explicitly by the lexicographer; Ajax also reordered the senses whenever the lexicographer opened an entry.

In the Set-of-Senses View, reordering was implemented with a simple numerical sort.

In the Complete Structure View, the implementation was complicated because we had to ensure that the reordered entry represented a legal entry structure; for instance, there could be no duplicate homograph or sense numbers. First we sorted the homographs and checked for duplicates. Then, within each homograph, we gathered all the senses together. We ensured that the hierarchy was complete. For instance, if the entry contained sense 3.1a we checked that it also contained senses 3 and 3.1, adding dummy senses if it didn't. Finally, we re-inserted the senses into the entry structure as indicated by their sense numbers.

This reordering was the one instance in the Complete Structure View where the functionality was not independent of the contents of the specification file. In this operation, Ajax knew that sense fields were special and that they could nest inside one another.

To renumber in the Complete Structure View, we simply traversed the entry hierarchy and assigned successive values at each level.

To renumber in the Set-of-Senses View, the challenge was to determine what structure was implied by the order of the senses. Our implementation started from the top and compared homograph and sense numbers for successive senses. We preserved the structural relationship implied by the original numbers; that is, if two successive senses appeared to belong to the same homograph in the original numbering, we assigned them to the same homograph in the new numbering. Similarly, we preserved the relationships between senses in a homograph. If sense 2a was a subsense of sense 2 in the original numbering, we ensured that it is still a subsense in the new numbering.

Figure 1 below shows how the homograph (Hom) and sense number fields look before and after renumbering. Fields originally without numbers get numbers assigned to them.

5.5 Communication between Ajax and Argus

As we described in section 4.1, the lexicographers used mnemonics to link corpus words with senses. Argus needed to convert those mnemonics into unambiguous UIDs: for example, if a lexicographer tagged a use of "leap" with the mnemonic `JUMPNOUN`, Argus needed to determine that the lexicographer meant the sense with UID 508118.

Insisting that all the mnemonics in the dictionary be unique would have placed an unreasonable burden on the lexicographers. Instead, we established the notion

Original		Renumbered	
Hom	Sense number	Hom	Sense number
1	6	1	1
1	5	1	2
empty	empty	2	1
empty	1	2	2
empty	1a	2	2a
1	3	3	1
empty	4	4	1
empty	7	4	2

Figure 1: Homograph and sense numbers before and after renumbering.

of an "active set" of mnemonics, and we required that there be no duplicates in the active set. In order to add mnemonics to the active set, the lexicographers had to ensure that the entry containing those mnemonics was in an Ajax window; then they pushed the "Tags" button in that window. The window's mnemonics remained active until they pushed the button again.

Ajax maintained a table of all active mnemonics from all its windows. Internally, Ajax stored the mnemonics in a normalized form: all uppercase characters, with no leading or trailing whitespace. When adding new mnemonics to the table, Ajax checked for duplicates. If it discovered a duplicate, it issued an error message and refused to activate the new mnemonics. Ajax did not check the legality of the mnemonic, however; illegal mnemonics were detected by Argus. Ideally, both Ajax and Argus should have checked their validity.

When the lexicographer activated a set of tags, Ajax sent Argus a list of "tag name, tag information" pairs. The tag information included the UID of the sense, its headword, its homograph number, and its sense number. (Argus needed the headword, homograph, and sense numbers to implement dictionary-order sorting.)

Ajax and Argus used the selection mechanism provided by the window manager to communicate information about active tags. The selection mechanism is normally used to implement cut-and-paste between the windows of different applications. Argus owned the selection, and Ajax wrote information to the selection owner, so Ajax pushed active tag information to Argus rather than having Argus pull it from Ajax.

If Argus started up after Ajax had already activated some tags, Argus would not know about the tags. In practice, we asked the lexicographers to turn the tags off and on again in this situation, which was rare.

We discovered that our assumption about when the lexicographers needed to use the mnemonics was too restrictive. There were times when the lexicographers wanted to use mnemonics in Argus without having the corresponding entry displayed in Ajax. If a search in Argus yielded some examples that were tagged with a UID that was not in the lexicographer's active set, Argus displayed only the numerical UID, not the mnemonic. When the lexicographers saw such tags, they were not always able to tell which entry to load into Ajax to activate the mnemonics. At other times, the lexicographers wanted to use mnemonics just so they could exclude collocates with a certain sense. In neither case did they need the entry in Ajax; they just wanted the mnemonics available to Argus.

5.6 Monitoring Active Tags

While editing an entry, the lexicographers could change information about an active tag, by changing the mnemonic itself or by changing the sense number associated with a mnemonic. Argus needed to know about these changes—but when? We didn't want to inform Argus on every keystroke: that would have been computationally expensive and also potentially incorrect, since the lexicographers might go through illegal states in the middle of their edit. For instance, all the tags within an entry had to be unique. However, if a lexicographer already had a tag "big", and wanted to add another tag "bigger", two tags "big" would temporarily exist before the lexicographer had typed the second "g" in "bigger".

The strategy we adopted to overcome this was to ask the window manager to signal the program whenever an active tag field gained or lost the input focus. (The window with the input focus is the window to which keystrokes will be sent.) We used the input focus as an indication that the lexicographer was beginning or ending an edit to a field.

When we gained the input focus for an active field, we recorded the value of that field. When we lost the input focus, we compared the new value to the old. If it had changed, we attempted to remove the old value from the active-tag table and add the new value. If this was successful, we informed Argus of the new tag information.

In general, this worked pretty well. The only drawback came when the change in input focus was explicit rather than implicit. If a lexicographer didn't need to make any other edits, the natural thing would be just to leave the input focus in the active field; it was odd to have to move the input focus explicitly in order to communicate with Argus.

There were other bookkeeping tasks connected with active tags. When adding or deleting a sense, it was important to check to see whether the tags in that entry were active, so that the tag could be added or removed from the tag table. There were a number of bugs that resulted in old tags being left in the active tag table, with the unfortunate result that Ajax would complain about duplicate tags when there were no duplicates visible.

Certain other operations, such as renumbering, changed some active fields without involving the focus mechanism. These operations had to cause the tags to be re-evaluated explicitly.

6 Conclusion

We have reported the systems decisions we made in building the software tools for the Hector project. In the course of the project, we formed some opinions that we present here.

After observing the painstaking care and effort it took the lexicographers to study the citations, to refine their thoughts about words and their meanings, and to write the dictionary entries, we realized that much of the lexicographic wisdom they acquired and recorded in the process might never appear in any printed dictionary. This seems to be both a pity and a loss to scholarship. We are convinced that this information should be stored as part of an on-line lexical database that would evolve as new information was added to it. Such a database, accessible using computer tools, could be an invaluable resource for compiling new dictionaries, and performing lexicographic research.

Moreover, we believe that this database and the tools for corpus lexicography could be built using commercially available software. This point was not at all obvious to us when we began the project. We built Hector practically from scratch. Some components were rewritten many times; others were discarded as our experience grew. At the end, we had a hand-crafted collection of software tools in very traditional areas: data compression, database storage and retrieval, a graphic user interface toolkit, a multi-user interactive application, and an assortment of free-standing utilities for analyzing text.

Having seen what worked and what didn't, what was really useful and what was interesting but not essential, we conclude that Version 2 of the tools would not be difficult to construct. Designing the software tools for corpus lexicography was hard, but having done so, we feel that the design could now be implemented with standard software packages. Furthermore, hardware components are now readily available with sufficient power and capacity to support a network-based system for indexing and searching.

In contrast, there are still no standard tools for natural language processing that are suitable for this work. For example, the syntactic analyzer that we used was better than one we could have built ourselves, but it was far from having the accuracy needed. Although the existing tools are adequate, in our opinion, improvement in this area would make the single biggest difference to corpus lexicography.

Acknowledgements

Thanks to our friends and colleagues who assisted us in designing and implementing the Hector system:

To the users of the Hector system, who had to educate us about the practical problems of lexicography and live with our evolving system: Sue Atkins, Katherine Barber, Peter Gilliver, Patrick Hanks, Helen Liebeck, Rosamund Moon, Della Thompson, and Bill Trumble.

To the Modula-3 crew, who fixed problems in the Modula-3 system and libraries, and extended the functionality of the language to meet our needs: Marc H. Brown, Steve Glassman, Bill Kalsow, Eric Muller, Mark Manasse, Greg Nelson, and Ted Wobber.

To Mike Burrows, for writing the Hector Index Server.

To Jim Horning, for keeping a watchful eye on the Hector project and providing wisdom at many critical points.

To our colleagues and co-implementors on the Hector project: Lucille Glassman, Cynthia Hibbard, and Mary-Claire van Leunen.

Thanks also to our friends and colleagues for their help with this report:

To Mary-Claire van Leunen for getting us started.

To Mike Burrows and Jim Horning for their constructive comments on an earlier version.

To Cynthia Hibbard and Lucille Glassman for their helpful suggestions on the report's presentation. Special thanks to Cynthia, without whose persistence and excellent editing this report would never have been written.

A Evaluating the Results

In the Hector overview paper[1], we listed things that the SRC Hector team hoped to accomplish by the end of the project. Here, we review the status of each of those items.

A.1 Better dictionary definitions

Goal: 500 dictionary entries.

At the beginning of the project, we selected 570 words as candidates for the database. The lexicographers decided to use some of these target words, and they included some other words that were not in the original target set. The exact words weren't critical, as long as they were chosen within the scope of the project's goals.

At the end of the project, the lexicographers had produced 1,510 dictionary entries. This far exceeded our original goal of 500; however, many of these were single-sense entries for compounds related to a broader headword.

For the lexicographers, the Hector experience reinforced the lessons of other corpus-based projects, like COBUILD[7]: a corpus is invaluable when working on the definition of ordinary words. Even within the limited scope of the Hector project, the lexicographers found a number of word senses that were overlooked in dictionaries not based on corpus evidence. For example, the metaphorical meaning of the word "capture", as in "capture two seats in Parliament", is missing from dictionaries such as Longman and the Oxford Concise, although it is present in corpus-based dictionaries such as COBUILD Student's and the American Heritage III. This sense accounts for 15% of the occurrences of "capture" in the corpus.

A.2 Links between the corpus and the senses

Goal: A 20-million-word corpus with 300,000 words linked to those 500 dictionary entries.

At the end of the project, the lexicographers had linked 230,847 words in the corpus to senses in 1,510 entries. This is 1.3% of the words in the corpus. Our original target—the number of occurrences of the original target words—was 366,670 links. Using this as a metric, the lexicographers accomplished 63% of what we thought they could do.

Clearly, we overestimated how much the lexicographers would be able to accomplish. Although there were delays that could be attributed to immature software and training time as new lexicographers learned to use the tools, we must conclude that we were overly optimistic about how much the computer would speed the basic task of deciding on sense-boundaries and establishing the links. The lexicographers still had to look at, and think about, every occurrence of each target word in the corpus.

A.3 Distribution of senses in the corpus

Goal: Statistical information on the distribution of words, wordclasses, and dictionary senses in the corpus.

We wrote a program called *entrystats* to calculate and display statistics about the distribution of a word within the corpus. It produced two kinds of statistics: one based on the set of senses that the lexicographer had defined in the dictionary entry, and one based on the wordclass tags that the lexicographer had assigned.

To calculate the distribution of senses of a headword, entrystats read its dictionary entry for information about the senses and any variant forms or variant spellings, and then produced statistical information on the frequency of each sense (and each variant, if any) in the corpus. The program ignored case variation in the corpus. Here is the output for the headword "treasure":

```
treasure:       13 per million

        80% noun (treasure 30%, treasures 49%)
        13% verb (treasure 4%, treasures 1%,
                  treasuring <1%, treasured 7%)
         7% adj  (treasured 7%)
- ------------------------------
  treasure
    1       thing         50%     112
    1.1     thingex       10%      22
    1.2     person         2%       4
    2       hoard          7%      15
    2a      mod            4%       8
    2.1     met            5%      12
    2.2     collection     1%       3
    3       wealth         2%       5
    4       cherish       13%      29
    4a      cherished      7%      16
```

The headword is followed by a count per million. The next three lines show the percentages for each wordclass, with the wordclass information further broken down into percentages for each wordform. The final section lists, for each sense in the entry, the sense number, the mnemonic assigned by the lexicographer, the percentage of all occurrences of the headword that were tagged with this sense, and the absolute count of occurrences in the corpus.

To compute the information on wordclass, we parsed the lexicographer's grammar field to extract a wordclass from the set {noun, verb, adverb, adjective, other}. We then generated the inflections for the wordclass, including expansions for all variant forms and variant spellings. If the lexicographer had defined a lexical form, we included that as well. (A lexical form is a sense that applies to only one wordform; for example, "treasured" is a lexical form for the adjective sense of "treasure".)

Using variant forms, variant spellings, case variations, wordclass inflections, and lexical forms gave us a list of all possible wordforms for a wordclass. When a wordform appeared in more than one wordclass (e.g., "treasures," which can be both a noun and a verb), we checked the tag in the corpus to do the disambiguation. We reported the frequency of inflected forms as a percentage of the relevant baseform.

A.4 The predictive value of the corpus links

Goal: A test of the predictive value of the links between words and senses.

One of the goals of the Hector project was to explore automatic sense disambiguation: could we use the computer to determine the dictionary sense of a word by examining its context? For example, can a computer determine the correct sense of "key" in the sentence, "This is the key difference in industrial performance"? Does it mean "something of vital importance" or "a system of notes in a piece of music"?

Initially we thought that we would have to log the search requests that the lexicographers made during their disambiguation to infer the context that characterized a sense. However, the lexicographers decided to describe their disambiguations explicitly, as they were doing the lexicography. They developed a notation called *clues* for describing conditions about the context of a word that could be used for disambiguation.

The clues could be lexical (collocations on individual words, lexical sets, or even punctuation), morphological, grammatical (proper noun, plural noun), or syntactic (subject, object). Clues could refer to the source word, to collocates, or to any word in the same clause.

Clues for nouns were typically just collocates; clues for attributive adjectives had the semantic class of the nouns they modified; clues for verbs were usually syntactic patterns. Our lexicographers concentrated on verbs and adjectives: there seemed little point in recording collocate information for nouns manually, since it was easy to derive significant collocates automatically. Ideally they would have created a clues field whenever two or more senses of a word shared the same wordclass; in practice, it was often impossible to capture distinctions sensibly.

Although we consider the clues mostly as the basis for future work, we did run some experiments to evaluate them. We evaluated a clue by translating it into an Argus query, running the query against the corpus, and counting how many of the query matches were tagged with the sense associated with that clue. The percentage with the correct tag was an indication of the effectiveness of that clue in identifying the sense. A clue with 90% effectiveness, for example, would be quite useful for disambiguation; a clue with 25% effectiveness would not.

This experiment was flawed in a number of ways. First, we were running our test over the corpus: we were evaluating the clues against the same data that was used to generate the clues in the first place. Second, the clues also expressed knowledge that could not be translated into a query; our inability to use all the information in the clues reduced the effectiveness of our queries. For example, many of the clues relied on a (mythical) very sophisticated parser; they equated active and passive voice, assumed understanding of anaphoric constructions like "Naomi likes a purple shirt better than a blue one", and so on. We reduced syntactic restrictions to wordclass restrictions; a word marked as the subject was probably a noun or pronoun.

Once the clues were evaluated for effectiveness, we attempted disambiguation. If we were disambiguating a word in a sentence, we first needed to determine which entries might possibly apply. Then we compared each clue in all these entries against the sentence, to see whether the sentence satisfied the clue. Finally, we chose the sense corresponding to the most accurate clue that the sentence satisfied.

As an example, suppose we were disambiguating "key" in the sentence, "This is the key difference in industrial performance", and we had the following clues:

1. "key" immediately followed by a noun.

2. "key" collocated with any word in the lexical set "vehicle".

3. "key" collocated with any of the words "major", "minor", "sharp", or "flat".

Of these clues, only the first is satisfied by this sentence. So we would choose the sense associated with it, "something of vital importance".

To test this approach, we disambiguated every entry the lexicographers produced. The results of this disambiguation varied wildly. Sometimes there was sufficient information available to distinguish between senses, sometimes not. Some of the lexicographers wrote clues that were too narrow; others wrote clues that were too broad. None of these problems was surprising, of course: it was hard for the lexicographers to do the disambiguations in the first place, by hand, and even harder for them to try to capture their understanding using an unfamiliar and untested notation.

If we consider only words with more than one or two senses, and more than 500 instances in the corpus, some of the more successfully disambiguated words include:

complain	87%	981 correct out of 1124	5 senses
spite	78%	453 correct out of 584	5 senses
cease	78%	413 correct out of 528	4 senses
vegetable	77%	501 correct out of 652	4 senses
key	76%	1590 correct out of 2096	35 senses

Some of the less successfully disambiguated words include:

pledge	10%	52 correct out of 505	14 senses
generous	13%	74 correct out of 561	7 senses
breed	12%	80 correct out of 679	25 senses
steam	23%	122 correct out of 541	31 senses
evil	24%	122 correct out of 507	18 senses
angle	25%	152 correct out of 600	20 senses
steer	32%	168 correct out of 523	13 senses

From these figures, we observed that most of the results were significantly better than random even for the less successfully disambiguated words.

We conjecture that it might be useful to use the entrystats information when doing disambiguation: if in doubt, we would choose the more common sense.

A.5 Evaluating automatic wordclass assignments

Goal: An evaluation of the automatic wordclass assignments.

A hand-tagged corpus is a useful testbed for evaluating computer programs that attempt to analyze natural language. By comparing the program's results with the information implied by the sense tags, we can evaluate the accuracy of the program against a large body of data.

We were interested in evaluating the accuracy of the wordclass assignments generated by Adam and the Houghton Mifflin parser. Originally, we had hoped the lexicographers would correct, or at least note, wordclass errors while they were sense-tagging entries; in reality, they didn't have time. However, we found we could detect wordclass errors ourselves by analyzing the grammar fields in the lexicographers' dictionary entries.

Generally, there is a set of wordclasses that are valid for any grammar field. For instance, if the grammar for a sense is "proper noun," then the wordclass assigned might be NPL (capitalized locative noun), NPLS (capitalized plural locative noun), NP (proper noun), NPS (plural proper noun), NPT (capitalized titular noun), or NPTS (capitalized plural titular noun).

We made an optimistic evaluation by checking whether the tag assigned to a word fell within the set of wordclasses for its sense's grammar. This was optimistic because the tagger might have assigned a wordclass that was wrong but that was still within the right set. For instance, it might have identified a singular proper noun as a plural proper noun.

For the portion of the corpus that was sense-tagged during the Hector project, both programs were just under 90% accurate. It would be premature to extrapolate this accuracy to the entire corpus, since the sample of words sense-tagged was not random.

For instance, we were surprised that neither program seemed to recognize cardinal numbers successfully until we noticed that the only word that had been tagged as a cardinal was "mill", meaning a million. We expect that the programs would be more successful with more common cardinals.

On the other hand, since the pilot project focused only on words that occurred a few thousand times in the corpus, we had no evaluation of the programs for words like "take" or "run" that both occur very frequently and have dozens of senses.

Although both programs had the same overall accuracy, they made different errors. Adam identified prepositions, adjectives, and nouns more successfully than the Houghton Mifflin parser did, but the Houghton Mifflin parser was more successful than Adam at identifying plural nouns, proper nouns, verbs, adverbs, and conjuncts.

A.6 Evaluation of corpus clean-up

Goal: Contrast the raw and the cleaned-up versions of the corpus.

Our original intention was to run the corpus search tools on both the raw and cleaned-up versions of the corpus, take measurements, and report our findings. By doing this we hoped to gain some statistical evidence that would either support or refute the thesis that extensive handwork on a corpus isn't worth doing.

As it turned out, we never made the measurements.

B The Adam wordclass tagger

This section describes some of the problems we faced in building Adam, a program that identified wordclasses automatically. The interesting aspect of this program was that although the core algorithm was quite simple to state, making it work on real-world text was not simple at all. The core algorithm in Adam had only 50 lines of code, but it was surrounded by over 4,000 lines of code that handled punctuation, abbreviations, sentence boundaries, numbers, and unknown words.

B.1 Optimizing the core

The core algorithm, as described by Ken Church[2], uses lexical probabilities (What are the odds that this word is a noun?), and contextual probabilities (What are the odds that a noun would follow a determiner and an adjective?). It is the product of those probabilities that determines the best tagging sequence. We used the frequency tables and the lexicon from the Lancaster-Oslo-Bergen (LOB) corpus[3]. A straightforward implementation of this algorithm, however, was far too slow; tagging the entire corpus would have taken weeks.

Fortunately, Jim Horning, a colleague at SRC, pointed out that if we were careful, we could get nearly the same behavior by multiplying the *frequencies*, which were in the original LOB data, rather than the *probabilities*, which had to be computed. Furthermore, instead of multiplying numbers, we could add their logarithms, which was faster. Finally, since the data was static, we could pre-compute all the logarithms, while we processed the original LOB data. We stored the logarithms in files that were loaded into tables in memory when Adam started up. Tagging the corpus became an overnight job.

That was the 50-line core of the code. The rest of this section discusses the problems we faced in dealing with the text.

B.2 Tags and Tokens

The tagger's basic function was to read an SGML-marked file and record a part-of-speech tag for each token in that file. But what is a token, and how do you find it in raw text? Words are tokens, but so are punctuation marks and SGML tags.

Some SGML symbols were treated as text. Some were delimiters and formed separate tokens (e.g., "&dash."). They were in the LOB lexicon, although written in a different notation; the LOB version of "&dash.", for example, was written "*-". Others were left as part of the token. For example, the French name André was written in SGML as "Andre&acute.". Our SGML parser treated that as two tokens: a word, "Andre", and a form, "&acute.". The tagger had to put them back together, since that name was in the LOB lexicon, written "Andre*?2".

Other SGML symbols were handled on a case-by-case basis. For example, {`fraction ...`} was simply treated as one token and was always tagged CD (cardinal) without further interpretation. The most complicated case was the form that marked typographical errors in the corpus: {`typo bad="..." good="..."`}. The "good" text was processed as if it were the normal input. It could contain closing-tags, as in {`typo bad="//s>" good="</s>"`}, and other unbalanced text.

B.3 Apostrophes: contractions and genitives

Following the conventions of the LOB data, contractions were represented as two tokens: "can't" was tagged as "can" (MD = modal) + "n't" (XNOT). This presented some difficulties when we were searching for words in the corpus; some "words" were actually adjacent tokens, not separated by whitespace or punctuation.

The LOB data included some genitives (the company's logo) but not many. The LOB wordclass-set included 27 genitive tags, 23 of which were for words that were orthographically marked with 's or s'. For example, NN represented a singular common noun (company), so NN$ represented a singular common noun + genitive (company's). Likewise, CD indicated a cardinal (0), so CD$ indicated cardinal + genitive (0's).

This data was unsatisfactory, for several reasons. First, there was no consistency of coverage (the lexicon contained "board's" but not "boarder's"). Second, none of the LOB words ending in 's were marked as contractions for "is", "has", or "us". (John's in New York. He's been there before. Let's join him.) Third, "0's" indicated a plural (How many 0's are there in one million?) as often as it was used as a genitive (Knuth discusses 0's origin). Indeed, the whole notion of putting individual numbers in the lexicon was problematic.

In the tagger, then, a final 's was always split off, and in a final s', the s stayed but the ' was split off. We deleted all 23 genitive tags (e.g., NN$). In the contextual-probability table, we replaced every occurrence of NN$ with NN and a new tag, $. The tag for 's was either $ (genitive), PP1OS (us), HVZ (has), or BEZ (is). The tag for ' itself was either $ (genitive) or, with *very* low probability, &FW (foreign word).

All the numbers were removed from the lexicon. The word-scanner recognized several forms of numbers lexically, and marked them all as CD (cardinals). These included integers (123456), real numbers (1234.567), and numbers with commas in the right places (1,234,567) optionally followed by two decimals (1.98 or 12,345.67 [for money]). Preceding plus-signs or minus-signs were included.

B.4 Abbreviations

Abbreviations at the end of a sentence presented a problem for the tagger: should the final period be marked as part of the abbreviation? We decided that final periods should be attached to the abbreviation, unless they also indicated the end of a sentence. For example, in "Toys, etc., are fun.", there was a token for "etc.", but in "I like toys, etc.", there was one token for an abbreviation spelled "etc" and another for the final ".". In both cases, the tag on the abbreviation was the same (RB = adverb).

B.5 Sentence boundaries

By far the most difficult problem for the tagger was detecting sentence boundaries in free text. The LOB corpus-markings included many cases where there was no punctuation at the end of a sentence; a sentence could end with a singular common noun or a past-tense verb, for example. Other sentences ended with commas; more precisely, some sentences *began* after a comma, as in "I'm lost, he thought". Since sentence-beginnings weren't marked in the Hector corpus, there was the possibility that a sentence could begin after *any* noun, verb, comma, etc.

Adam considered all of these possibilities, and used the frequencies from the LOB data to determine the most likely one. A standard method of handling optional tags (marking the "invisible" sentence-beginnings) would be for the tagger to consider tagging-sequences of different lengths. After every noun, for example, we could have considered two possible tagging-sequences, one ending in NN, and the other ending in NN followed by ^ (the tag for sentence-beginnings). In fact, the early versions of the tagger did just this.

However, there was some overhead in managing sequences of different length. Moreover, the optimizations in the core algorithm required that all the alternative paths have the same length. So we invented *pseudo-tags* or *boundary tags* with names like NN^, which indicated a singular common noun at the end of a sentence. The most popular boundary tag indicated a period at the end of a sentence. We estimated the frequencies for these new tags.

B.6 Unknown words

The LOB lexicon included 45,561 different words; while that gave Adam excellent coverage of the basic English vocabulary, there were many words in the corpus that weren't in the lexicon, especially proper nouns. In order to make reasonable guesses about the wordclass tags for unknown words, we applied some simple heuristics.

When we processed each word, we looked it up in the LOB lexicon. For the word "set" for example, the lexicon indicated that it could be a verb, a past participle, a past-tense verb, a noun, and an adjective, with various frequencies. Each of those possibilities was considered, that is, added to the set of lexical probabilities for the word.

If a word began with a capital letter, it may or may not have been in the LOB lexicon; "Meehan" was, for example, but "Atkins" wasn't. If it wasn't in the lexicon, or if it was the first word in the sentence, we also looked up its lower-case version, and considered that possibility as well.

If we still hadn't found anything, we applied other heuristics. First, if the word ended in "ly", we guessed that it was an adverb. (Here, "guessed" means that we treated it as if it actually occurred, *exactly once*, in the LOB lexicon.) This was a good rule, although there were many exceptions; "curly" is not an adverb, and "doubtless" is.

If it ended in "ed", we guessed that it was a past-tense verb or a past participle. If it ended in "ing", we guessed that it was a present participle.

If it ended in "er", we guessed that it was a comparative adjective; if it ended in "est", we guessed that it was a superlative adjective.

In addition to those heuristics, we used one final set of rules. In practice, there seemed to be only three reasonable guesses for unknown words: nouns, verbs, and adjectives. Many of the other classes were closed sets; all the prepositions, for example, were already in the lexicon, so there was no point in guessing that an unknown word might be a preposition.

If the unknown word did not have an initial capital letter, or if it did but it was at the beginning of the sentence, we guessed that it might be an adjective (JJ), a verb (VB), and either a singular common noun (NN) or a plural common noun (NNS), depending on whether it ended with an s.

If the word did begin with a capital letter, regardless of whether it was at the beginning of a sentence, we guessed that it could be a proper adjective (JNP). If it ended with an s, we guessed that it might be a plural proper noun (NPS, e.g., "Rockefellers") or a capitalized plural common noun (NNPS, e.g., "Californians"). If it didn't end with s, we guessed that it might be a proper noun (NP, "Rockefeller") or a capitalized common noun (NNP, "Californian").

Thus, the unknown word “Meese” was usually tagged as a proper noun, but in the sentence “Meese ordered Jenkins to prepare a memo”, it was tagged as a singular common noun, while in the sentence “After he made his pitch, Meese ordered Jenkins to prepare a memo”, it was tagged as an adjective. (So was “ordered”, which is a known word.)

B.7 Accuracy

No matter how much tuning one does, a stochastic tagger like Adam is ultimately only as good as the data that drives it. In the LOB corpus, the word “a” was tagged as an article 21,926 times, but it was also tagged as a post-determiner, a preposition, a simple adjective, an adverb, a foreign word, and a letter of the alphabet. Certainly there are tricky uses of this word (“3 times a year”, “many a fool has tried this”), and some of the taggings (e.g., simple adjective) were probably wrong. But we couldn't just go in and fix the data, willy-nilly, changing “a” so that it was *always* tagged as an article, for example. The more changes you make, the less reliable your results are.

When we made changes to the LOB data, we tried to preserve the original information exactly. The heuristics for dealing with unknown words, for example, were neither consistent nor inconsistent with the LOB data, although a simple scan of the LOB wordlist reveals that most of the words in the dictionary are nouns, so our heuristics were at least reasonable.

References

[1] Lucille Glassman, Dennis Grinberg, Cynthia Hibbard, James Meehan, Loretta Guarino Reid, and Mary-Claire van Leunen. *Hector: Connecting Words with Definitions*, Digital Equipment Corporation Systems Research Center Report 92A, October, 1992.

An accompanying videotape, Report 92B, demonstrates the Hector tools.

Also published in the *Proceedings of the Eighth Annual Conference of the UW Centre for the New OED and Text Research*, University of Waterloo, Canada, 1992.

[2] Ken Church. A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the Second Conference on Applied Natural Language Processing*, Austin, Texas, 1988.

[3] Stig Johansson and Knut Hofland. *Frequency Analysis of English Vocabulary and Grammar*. Oxford University Press, 1989.

[4] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[5] Walter Tichy. RCS—A System for Version Control. In *Software Practice and Experience*, July, 1985.

[6] Martin Bryan. *SGML: An Author's Guide to the Standard Generalized Markup Language*. Addison-Wesley Publishing Co., 1988.

[7] J. M. Sinclair, editor. *Looking Up: An acount of the COBUILD Project in lexical computing*. Collins, 1987.